散列查找

目录

[散列查找 1](#_Toc450588021)

[平衡树AVL 2](#_Toc450588022)

[红黑书 6](#_Toc450588023)

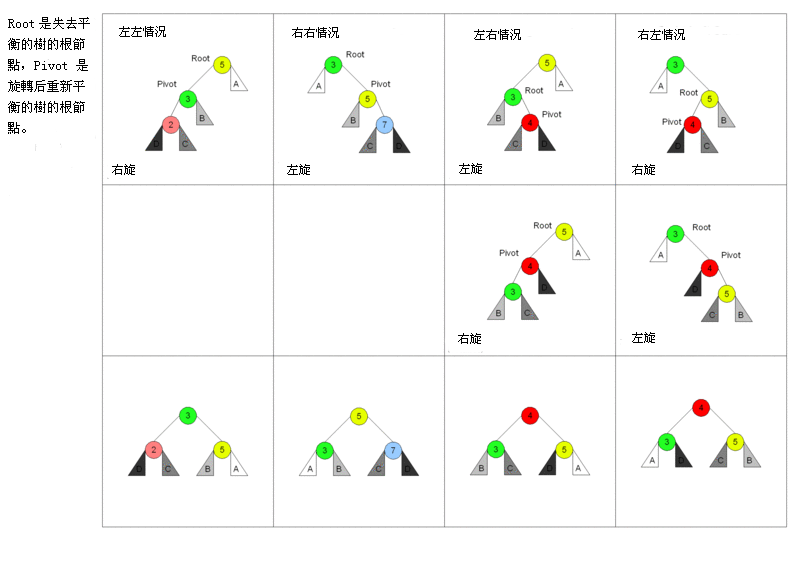
[散列查找(HASH TABLE) 13](#_Toc450588024)

## 平衡树AVL

在计算机科学中，AVL树是最先发明的自平衡二叉查找树。在AVL树中任何节点的两个子树的高度最大差别为一，所以它也被称为高度平衡树。查找、插入和删除在平均和最坏情况下都是O（log n）。增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。

节点的平衡因子是它的左子树的高度减去它的右子树的高度（有时相反）。带有平衡因子1、0或 -1的节点被认为是平衡的。带有平衡因子 -2或2的节点被认为是不平衡的，并需要重新平衡这个树。平衡因子可以直接存储在每个节点中，或从可能存储在节点中的子树高度计算出来。

旋转操作：



插入

向AVL树插入，可以透过如同它是未平衡的二叉查找树一样，把给定的值插入树中，接着自底往上向根节点折回，于在插入期间成为不平衡的所有节点上进行旋转来完成。因为折回到根节点的路途上最多有1.44乘log n个节点，而每次AVL旋转都耗费固定的时间，所以插入处理在整体上的耗费为O(log n) 时间。

删除

从AVL树中删除，可以透过把要删除的节点向下旋转成一个叶子节点，接着直接移除这个叶子节点来完成。因为在旋转成叶子节点期间最多有log n个节点被旋转，而每次AVL旋转耗费固定的时间，所以删除处理在整体上耗费O(log n) 时间。

搜寻

可以像普通二叉查找树一样的进行，所以耗费O(log n)时间，因为AVL树总是保持平衡的。不需要特殊的准备，树的结构不会由于查找而改变。

代码如下：

#include <bits/stdc++.h>

using namespace std;

class AVL

{

public:

struct Node

{

int data, height;

Node \*lson, \*rson;

void show() { printf("%d(%d,%d):|%d|\n", data, lson? lson->data:0,rson? rson->data:0, height); }

Node(){}

Node(int d):data(d),height(1),lson(NULL),rson(NULL){}

};

Node \*root;

AVL() { root = NULL; }

void show(Node \*now)

{

if(now == NULL) return;

show(now->lson);

now->show();

show(now->rson);

}

void update\_height(Node \*r){ r->height = max(r->lson==NULL? 0:r->lson->height, r->rson==NULL? 0:r->rson->height) + 1; }

int get\_height(Node \*r){return r == NULL? 0:r->height;}

void rotate\_right(Node \*Ro, Node \*Pi)

{

Node \*tmp = Pi->lson;

Pi->lson = Pi->rson;

Pi->rson = Ro->rson;

Ro->rson = Ro->lson;

Ro->lson = tmp;

swap(Ro->data, Pi->data);

update\_height(Pi);

update\_height(Ro);

}

void rotate\_left(Node \*Ro, Node \*Pi)

{

Node \*tmp = Pi->rson;

Pi->rson = Pi->lson;

Pi->lson = Ro->lson;

Ro->lson = Ro->rson;

Ro->rson = tmp;

swap(Ro->data, Pi->data);

update\_height(Pi);

update\_height(Ro);

}

void rotate(Node \*now)

{

update\_height(now);

if(abs(get\_height(now->lson) - get\_height(now->rson)) <= 1) return;

int b1 = get\_height(now->lson) - get\_height(now->rson);

int b2 = b1 > 0? (get\_height(now->lson->lson) - get\_height(now->lson->rson)):(get\_height(now->rson->lson) - get\_height(now->rson->rson));

if(b1 > 0 && b2 >= 0) rotate\_right(now, now->lson);

else if(b1 < 0 && b2 <= 0) rotate\_left(now, now->rson);

else if(b1 > 0 && b2 <= 0)

{

rotate\_left(now->lson, now->lson->rson);

rotate\_right(now, now->lson);

}

else

{

rotate\_right(now->rson, now->rson->lson);

rotate\_left(now, now->rson);

}

}

void insert(Node \*&now, int d)

{

if(now == NULL)

{

now = new Node(d);

return ;

}

if(now->data == d) return;

if(now->data > d)

insert(now->lson, d);

else

insert(now->rson, d);

rotate(now);

}

bool search(Node \*now, int d) {return now == NULL? false:(now->data == d? true:(now->data > d? search(now->lson,d):search(now->rson,d)));}

int getMinData(Node \*now)

{

while(now->lson) now = now->lson;

return now->data;

}

void del(Node \*&now, int d)

{

if (now == NULL) return;

if (d == now->data)

{

if (now->lson && now->rson)

{

now->data = getMinData(now->rson);

del(now->rson, now->data);

}

else if (now->lson != NULL)

now = now->lson;

else if (now->rson != NULL)

now = now->rson;

else

now = NULL;

}

else if (d < now->data) del(now->lson, d);

else del(now->rson, d);

if (now != NULL) rotate(now);

}

bool del\_data(int d)

{

if(!search(root, d)) return false;

del(root, d);

return true;

}

}avl;

int main()

{

int x;

for (int i = 0; i < 50; i++)

{

x = rand() % 50;

// scanf("%d", &x);

avl.insert(avl.root, x);

}

cout << "=========" << endl;

avl.show(avl.root);

cout << "=========" << endl;

while(true)

{

scanf("%d", &x);

printf("%s\n", avl.del\_data(x)? "deleted":"not found");

cout << "=========" << endl;

avl.show(avl.root);

cout << "=========" << endl;

}

return 0;

}

/\*

1 2 3 4 5 6 7 8 9 10

\*/

## 红黑书

红黑树（英语：Red–black tree）是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。它是在1972年由鲁道夫·贝尔发明的，他称之为"对称二叉B树"，它现代的名字是在Leo J. Guibas和Robert Sedgewick于1978年写的一篇论文中获得的。它是复杂的，但它的操作有着良好的最坏情况运行时间，并且在实践中是高效的：它可以在O(log n)时间内做查找，插入和删除，这里的n是树中元素的数目。

性质

红黑树是每个节点都带有颜色属性的二叉查找树，颜色为红色或黑色。在二叉查找树强制一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求：

节点是红色或黑色。

根是黑色。

所有叶子都是黑色（叶子是NIL节点）。

每个红色节点必须有两个黑色的子节点。（从每个叶子到根的所有路径上不能有两个连续的红色节点。）

从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。

插入

我们首先以二叉查找树的方法增加节点并标记它为红色。（如果设为黑色，就会导致根到叶子的路径上有一条路上，多一个额外的黑节点，这个是很难调整的。但是设为红色节点后，可能会导致出现两个连续红色节点的冲突，那么可以通过颜色调换（color flips）和树旋转来调整。）下面要进行什么操作取决于其他临近节点的颜色。同人类的家族树中一样，我们将使用术语叔父节点来指一个节点的父节点的兄弟节点。注意：

性质1和性质3总是保持着。

性质4只在增加红色节点、重绘黑色节点为红色，或做旋转时受到威胁。

性质5只在增加黑色节点、重绘红色节点为黑色，或做旋转时受到威胁。

**情形1**:新节点N位于树的根上，没有父节点。在这种情形下，我们把它重绘为黑色以满足性质2。因为它在每个路径上对黑节点数目增加一，性质5符合。

**情形2**:新节点的父节点P是黑色，所以性质4没有失效（新节点是红色的）。在这种情形下，树仍是有效的。性质5也未受到威胁，尽管新节点N有两个黑色叶子子节点；但由于新节点N是红色，通过它的每个子节点的路径就都有同通过它所取代的黑色的叶子的路径同样数目的黑色节点，所以依然满足这个性质。

**情形3**:如果父节点P和叔父节点U二者都是红色，（此时新插入节点N做为P的左子节点或右子节点都属于情形3，这里右图仅显示N做为P左子的情形）则我们可以将它们两个重绘为黑色并重绘祖父节点G为红色（用来保持性质5）。现在我们的新节点N有了一个黑色的父节点P。因为通过父节点P或叔父节点U的任何路径都必定通过祖父节点G，在这些路径上的黑节点数目没有改变。但是，红色的祖父节点G可能是根节点，这就违反了性质2，也有可能祖父节点G的父节点是红色的，这就违反了性质4。为了解决这个问题，我们在祖父节点G上递归地进行**情形1**的整个过程。（把G当成是新加入的节点进行各种情形的检查）

**情形4**:父节点P是红色而叔父节点U是黑色或缺少，并且新节点N是其父节点P的右子节点而父节点P又是其父节点的左子节点。在这种情形下，我们进行一次左旋转调换新节点和其父节点的角色;接着，我们按**情形5**处理以前的父节点P以解决仍然失效的性质4。注意这个改变会导致某些路径通过它们以前不通过的新节点N（比如图中1号叶子节点）或不通过节点P（比如图中3号叶子节点），但由于这两个节点都是红色的，所以性质5仍有效。

**情形5**：父节点P是红色而叔父节点U是黑色或缺少，新节点N是其父节点的左子节点，而父节点P又是其父节点G的左子节点。在这种情形下，我们进行针对祖父节点G的一次右旋转；在旋转产生的树中，以前的父节点P现在是新节点N和以前的祖父节点G的父节点。我们知道以前的祖父节点G是黑色，否则父节点P就不可能是红色（如果P和G都是红色就违反了性质4，所以G必须是黑色）。我们切换以前的父节点P和祖父节点G的颜色，结果的树满足性质4。性质5也仍然保持满足，因为通过这三个节点中任何一个的所有路径以前都通过祖父节点G，现在它们都通过以前的父节点P。在各自的情形下，这都是三个节点中唯一的黑色节点。

代码如下：

void insert(const T& data)

{

ptr y = nul, \*x = &root;

//find the place to insert

while(\*x != nul)

{

y = \*x;

if (data < (\*x)->data)

x = &((\*x)->lc);

else

x = &((\*x)->rc);

}

//insert

\*x = new Node(data, y, RED);

ptr p = \*x;

p->lc = p->rc = nul;

// adjust

ptr uncle, pa, gpa;

while (p->pre->color == RED)

{

pa = p->pre;

gpa = p->pre->pre;

if (gpa->lc == pa)

{

uncle = gpa->rc;

if (uncle->color == RED)

{

gpa->color = RED;

uncle->color = BLACK;

pa->color = BLACK;

p = gpa;

continue;

}

if (pa->rc == p)

{

rotate\_left(p);

p = p->lc;

}

p->pre->color = BLACK;

p->pre->pre->color = RED;

rotate\_right(p->pre);

}

else

{

uncle = p->pre->pre->lc;

if (uncle->color == RED)

{

gpa->color = RED;

uncle->color = BLACK;

pa->color = BLACK;

p = gpa;

continue;

}

if (pa->lc == p)

{

rotate\_right(p);

p = p->rc;

}

p->pre->color = BLACK;

p->pre->pre->color = RED;

rotate\_left(p->pre);

}

}

if (p->pre == nul)

{

p->color = BLACK;

root = p;

return;

}

}

删除

如果需要删除的节点有两个儿子，那么问题可以被转化成删除另一个只有一个儿子的节点的问题（为了表述方便，这里所指的儿子，为非叶子节点的儿子）。对于二叉查找树，在删除带有两个非叶子儿子的节点的时候，我们找到要么在它的左子树中的最大元素、要么在它的右子树中的最小元素，并把它的值转移到要删除的节点中（如在这里所展示的那样）。我们接着删除我们从中复制出值的那个节点，它必定有少于两个非叶子的儿子。因为只是复制了一个值，不违反任何性质，这就把问题简化为如何删除最多有一个儿子的节点的问题。它不关心这个节点是最初要删除的节点还是我们从中复制出值的那个节点。

在本文余下的部分中，我们只需要讨论删除只有一个儿子的节点（如果它两个儿子都为空，即均为叶子，我们任意将其中一个看作它的儿子）。如果我们删除一个红色节点（此时该节点的儿子将都为叶子节点），它的父亲和儿子一定是黑色的。所以我们可以简单的用它的黑色儿子替换它，并不会破坏性质3和性质4。通过被删除节点的所有路径只是少了一个红色节点，这样可以继续保证性质5。另一种简单情况是在被删除节点是黑色而它的儿子是红色的时候。如果只是去除这个黑色节点，用它的红色儿子顶替上来的话，会破坏性质5，但是如果我们重绘它的儿子为黑色，则曾经通过它的所有路径将通过它的黑色儿子，这样可以继续保持性质5。

需要进一步讨论的是在要删除的节点和它的儿子二者都是黑色的时候，这是一种复杂的情况。我们首先把要删除的节点替换为它的儿子。出于方便，称呼这个儿子为N（在新的位置上），称呼它的兄弟（它父亲的另一个儿子）为S。在下面的示意图中，我们还是使用P称呼N的父亲，SL称呼S的左儿子，SR称呼S的右儿子。我们将使用下述函数找到兄弟节点：

**情形1:** N是新的根。在这种情形下，我们就做完了。我们从所有路径去除了一个黑色节点，而新根是黑色的，所以性质都保持着。

**情形2：** S是红色。在这种情形下我们在N的父亲上做左旋转，把红色兄弟转换成N的祖父，我们接着对调N的父亲和祖父的颜色。完成这两个操作后，尽管所有路径上黑色节点的数目没有改变，但现在N有了一个黑色的兄弟和一个红色的父亲（它的新兄弟是黑色因为它是红色S的一个儿子），所以我们可以接下去按**情形4**、**情形5**或**情形6**来处理。

**情形3：** N的父亲、S和S的儿子都是黑色的。在这种情形下，我们简单的重绘S为红色。结果是通过S的所有路径，它们就是以前*不*通过N的那些路径，都少了一个黑色节点。因为删除N的初始的父亲使通过N的所有路径少了一个黑色节点，这使事情都平衡了起来。但是，通过P的所有路径现在比不通过P的路径少了一个黑色节点，所以仍然违反性质5。要修正这个问题，我们要从**情形1**开始，在P上做重新平衡处理。

**情形4：** S和S的儿子都是黑色，但是N的父亲是红色。在这种情形下，我们简单的交换N的兄弟和父亲的颜色。这不影响不通过N的路径的黑色节点的数目，但是它在通过N的路径上对黑色节点数目增加了一，添补了在这些路径上删除的黑色节点。

**情形5：** S是黑色，S的左儿子是红色，S的右儿子是黑色，而N是它父亲的左儿子。在这种情形下我们在S上做右旋转，这样S的左儿子成为S的父亲和N的新兄弟。我们接着交换S和它的新父亲的颜色。所有路径仍有同样数目的黑色节点，但是现在N有了一个黑色兄弟，他的右儿子是红色的，所以我们进入了**情形6**。N和它的父亲都不受这个变换的影响。

**情形6：** S是黑色，S的右儿子是红色，而N是它父亲的左儿子。在这种情形下我们在N的父亲上做左旋转，这样S成为N的父亲（P）和S的右儿子的父亲。我们接着交换N的父亲和S的颜色，并使S的右儿子为黑色。子树在它的根上的仍是同样的颜色，所以性质3没有被违反。但是，N现在增加了一个黑色祖先：要么N的父亲变成黑色，要么它是黑色而S被增加为一个黑色祖父。所以，通过N的路径都增加了一个黑色节点。

此时，如果一个路径不通过N，则有两种可能性：

* 它通过N的新兄弟。那么它以前和现在都必定通过S和N的父亲，而它们只是交换了颜色。所以路径保持了同样数目的黑色节点。
* 它通过N的新叔父，S的右儿子。那么它以前通过S、S的父亲和S的右儿子，但是现在只通过S，它被假定为它以前的父亲的颜色，和S的右儿子，它被从红色改变为黑色。合成效果是这个路径通过了同样数目的黑色节点。

在任何情况下，在这些路径上的黑色节点数目都没有改变。所以我们恢复了性质4。在示意图中的白色节点可以是红色或黑色，但是在变换前后都必须指定相同的颜色。

代码如下：

bool remove(T data)

{

ptr \*x = &root;

while(\*x != nul)

{

if((\*x)->data == data)

{

//需要删除的节点有两个孩子节点，将该节点与其左子树中最大的节点交换

if((\*x)->lc != nul && (\*x)->rc != nul)

{

ptr tmp = (\*x)->lc;

while (tmp->rc != nul)

tmp = tmp->rc;

data = (\*x)->data = tmp->data;

x = &((\*x)->lc);

}

else

{

//三种简单情况的处理

if ((\*x)->lc->color == RED)

{

(\*x)->lc->pre = \*x;

\*x = (\*x)->lc;

(\*x)->color = BLACK;

return true;

}

if ((\*x)->rc->color == BLACK)

{

(\*x)->rc->pre = \*x;

\*x = (\*x)->rc;

(\*x)->color = BLACK;

return true;

}

if ((\*x)->color == RED)

{

\*x = nul;

return true;

}

//adjust

ptr pre = (\*x)->pre;

if ((\*x)->lc != nul)

{

(\*x)->lc->pre = \*x;

\*x = (\*x)->lc;

}

else if ((\*x)->rc != nul)

{

(\*x)->rc->pre = \*x;

\*x = (\*x)->rc;

}

else \*x = nul;

ptr p = \*x;

while (p != root && p->color == BLACK)

{

if (pre->lc == p)

{

ptr tmp = pre->rc;

if (tmp->color == RED)

{

tmp->color = BLACK;

pre->color = RED;

rotate\_left(tmp);

tmp = pre->rc;

}

if (tmp->lc->color == BLACK && tmp->rc->color == BLACK)

{

tmp->color = RED;

p = pre;

pre = p->pre;

}

else

{

if (tmp->lc->color == RED && tmp->rc->color == BLACK)

{

tmp->lc->color = BLACK;

tmp->color = RED;

rotate\_right(tmp->lc);

tmp = pre->rc;

}

tmp->color = pre->color;

pre->color = BLACK;

tmp->rc->color = BLACK;

rotate\_left(tmp);

return true;

}

}

else

{

ptr tmp = pre->lc;

if (tmp->color == RED)

{

tmp->color = BLACK;

pre->color = RED;

rotate\_right(tmp);

tmp = pre->lc;

}

if (tmp->rc->color == BLACK && tmp->lc->color == BLACK)

{

tmp->color = RED;

p = pre;

pre = p->pre;

}

else

{

if (tmp->rc->color == RED && tmp->lc->color == BLACK)

{

tmp->rc->color = BLACK;

tmp->color = RED;

rotate\_left(tmp->rc);

tmp = pre->lc;

}

tmp->color = pre->color;

pre->color = BLACK;

tmp->lc->color = BLACK;

rotate\_right(tmp);

return true;

}

}

}

}

}

else if (data < (\*x)->data)

x = &((\*x)->lc);

else

x = &((\*x)->rc);

}

return false;

}

## 散列查找(HASH TABLE)

hash表的在处理冲突的时候有两种决策，一种是开放寻址法，一种是挂链表的方法。

开放寻址法：

class HashTableOpen : public HashTableInterface

{

private:

string tb[mod];

public:

int insert(const string &str)

{

int h = BKDRHash(str);

while(tb[h] != "")

if(++h >= mod)

h = 0;

tb[h] = str;

return h;

}

int query(const string &str)

{

int h = BKDRHash(str);

while(tb[h] != "" && tb[h] != str)

if(++h >= mod)

h = 0;

if(tb[h] == "") return -1;

return h;

}

void clear()

{

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

tb[i].clear();

}

HashTableOpen(){clear();}

};

挂链表法:

class HashTableClose : public HashTableInterface

{

private:

vector<string> tb[mod];

public:

void clear()

{

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

tb[i].clear();

}

HashTableClose(){ clear(); }

int insert(const string& str)

{

int h = BKDRHash(str);

tb[h].push\_back(str);

return h;

}

int query(const string &str)

{

int h = BKDRHash(str);

for (int i = 0, len = tb[h].size(); i < len; i++)

if(tb[h][i] == str) return h;

return -1;

}

};

查询删除和插入的复杂度都近似为O(n)。