红黑树的建立及其应用

白云鹏
（中国地质大学（北京）信息工程学院 计算机科学与技术 1004161221）

摘要：二叉树，在大数据面前展现出了快速的查找方式。然而，二叉树有着一个致命的缺陷，那就是在数据有序的情况下，查找会变得低效。为了避免这个最坏的情况，基于二叉树的红黑树产生了。

Abstract: Binary-tree has shown a quick way to find data in the condition of large data. However, there is still one fatal flaw. If the data are sorted, the search operation will be inefficient. In order to avoid the condition. Red-black tree produced.

关键词：红黑树，二叉树，建立，分析，应用

0. 引言

在这快节奏的生活之中，算法方面也在不断追求着更快，快速的查找在实际应用中愈发显得更为重要，而在二叉树之上的红黑树，展现出了比普通二叉树更为优秀的快速查找。

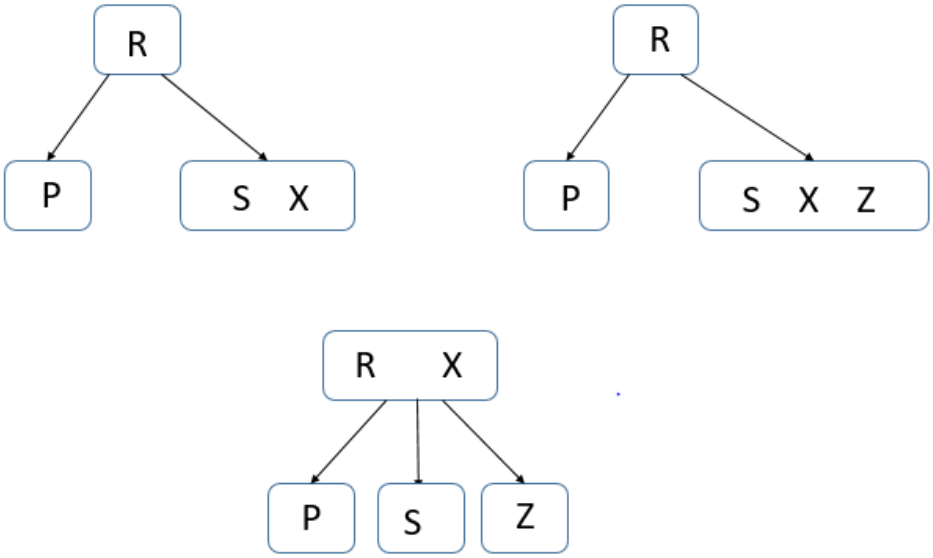
1. 红黑树的建立

首先，作为红黑树建立的基础，先介绍一下2-3树的生长规则。

2-3树是指将节点分为两种：2节点和3节点[1]。

2节点是指普通二叉树的节点。3节点就是指一个节点有两个元素，当然，这只是抽象的来讲。在插入新的元素时，如果插入位置的父节点是一个2节点，我们直接将二节点变为3节点，插入完成。如果插入位置的父节点是一个3-节点，如图，我们先将这个3节点变为一个4节点，然后将这个4节点的父节点变为一个3节点，当然，如果这个四节点的父节点原本就是一个3节点，我们就需要不断的重复这个操作直到父节点为2节点或者知道根节点。

如果到达了根节点，那么接下去，树的高度就会增加，这也是红黑树的生长规则。

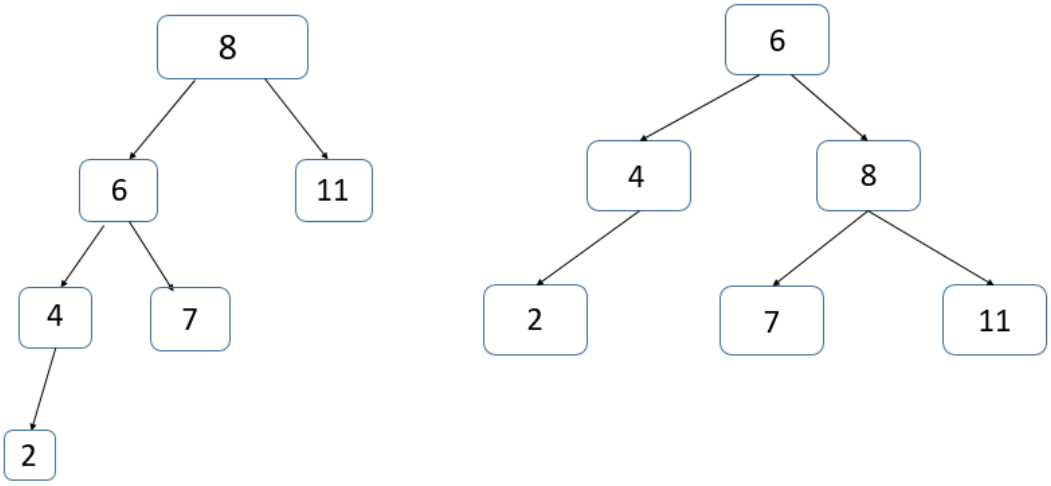


2-3树的插入

然而，要实现红黑树，更重要的是树的旋转，这也是平衡树的核心。在 AVL 树中，任意的两个

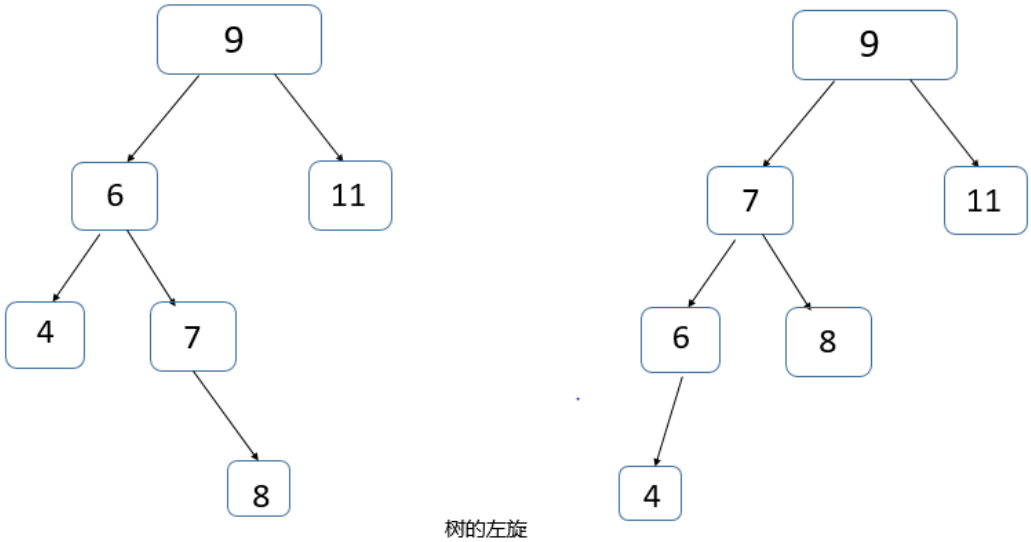
叶节点之间的高度超过1时，就要进行旋转。根据插入位置的不同，有着四种情况，其中两两对称，所以根本上只有两种情况。所以，在两种情况中只摘取了一种情况来讨论。

首先是比较简单的右旋，只需要一次旋转，它发生在新节点的插入位置在左子树的左儿子上。



树的右旋

相对的，另一种情况发生在插入位置在左子树的右儿子上。首先我们做一次左旋转，接着就回到来了上面的情况。



树的左旋

红黑树则综合了平衡二叉树以及2-3树的特性，有着下面的规律[2]：

1. 对于一个节点，它的颜色非黑即红。
2. 该节点与它的父节点或者子节点不能同时为红。
3. 任意的空节点到根节点路径上的黑色节点总数相同。
4. 根节点和空节点为黑色。

由此，便可得出：新增节点数必为黑色。因为对于一颗原来就是红黑树的树来说，新增一个节点，如果为黑，那么这小路径上的黑色节点数必然会比其他路径多出一个，所以新增的节点必须为红。

接着，就要开始建立红黑树了。

首先，对于一个节点，先按照二叉查找树的规则，找到它的插入位置。并记录下它的父节点。如果它的父节点为黑色，那么直接插入是不会影响整棵树的平衡性的。这时候直接返回，插入就成功了。

但是，如果父节点是红色的，事情就变得麻烦了。

根据伯父节点的颜色，分为了两大种情况。

如果伯父节点为红色，那么，和2-3树类似，不断地向上形成新的3节点或者直到根节点，完成了插入。

如果伯父节点为黑色，当然，根据规律4，如果伯父节点是空的，那也是黑色的。接着，由插入位置的不同，又分为了4种情况，其中两两对称。类似于二叉平衡树，如果插入位置的父节点是左子树并且插入位置也是父节点的右儿子，那么进行一次右旋转，否则先进行一次左旋转，再进行一次右旋转。同样，另一侧也是相对的。

至此，红黑树的插入已经结束了。

2.红黑树的优势分析

2.1 红黑树的中序遍历

2.2 红黑树的深度比较

2.3 红黑树的时间比较

红黑树建立在二叉树之上，因此，也拥有着二叉树的基本性质。

举个最简单的例子:二叉树的中序遍历，仍旧保持有序,图中使用了1000个随机数建立红黑树并进行中序遍历输出。

红黑树的中序遍历

但是，红黑树更多的是在于它与二叉树的不同之处。

二叉树的缺点在于数据有序时，会产生一个最坏的结果。而红黑树则是尽量避免这个结果。在根据一个有序的数组建树时，总有下面的式子成立(根结点处高度为零)：

Height(*binarytree*)/2+1>=Height(*rbtree*)>=Height(*binarytree*)/2

当节点数为1000时，红黑树深度为500，二叉树深度999.

当节点数为32765时，红黑树深度为16382，二叉树深度为32764.

……

这就表明红黑树在处理大数据上拥有着更强的普遍适用性，这是二叉树所不具备的。

而在一般情况下，红黑数的高度也是更加不可思议的。

对于32765个随机数的插入，如图是两种树的深度比较。

type	height 1	height 2	height 3	height 4	height 5	height 6	average
rbtree	16	16	17	15	15	16	15.83333
binarytree	63	72	68	69	65	67	67.33333

红黑树与二叉树的深度比较

可见红黑数在普遍情况下也要优于普通的二叉树。 最坏情况下，二叉树的插入时间复杂度是 O（N），而红黑树由于不断地在调整树形，复杂度只有 O（lg　*N*）。在这台电脑上，插入(1~32765)

这些数字，以下是所用时间：

type	time 1	time 2	time 3	time 4	time 5	time 6	average
binarytree	6.226	6.464	6.417	6.334	6.293	6.439	6.362167
rbtree	2.487	2.415	2.521	2.449	2.543	2.442	2.476167

红黑树与二叉树的插入时间比较

可见，红黑树在最坏情况下的插入更加高效，这也是因为在插入过程中不断调整，从而避免了最坏的情况。

在查询方面，由于红黑数更为漂亮的树形，所以时间复杂度也是低于二叉树的复杂度的，只需要 O（lg *N*）的复杂度。

3．红黑树的应用

3.1 桶排序

3.2 二维数组

3.3 prim 算法

虽然红黑树有着很优秀的性能，可是在实现方面确实是有一定的难度，主要就是在旋转操作以及节点颜色的分析方面。对于删除操作次数比较少的情况，可能会更倾向于选择 AVL 树来实现，但是，这样伟大的结构必然不可能被抛弃。在 C++中，map 以及 set 就是应用红黑树来储存数据的。而 map 作为重要的关联容器，更是发挥着重要的作用。那么下面直接拿 map 来阐述红黑树的应用。

(1)桶排序

传统的桶排序需要申请一个最大数值大小的空间，并且在没有进行离散化的前提下，也只能对整型或者是字符类型进行排序。在使用了 map 之后，申请的个数也只是不同元素的个数，而且也支持了各种有默认比较运算符的类型。

```
#include<iostream>
#include<map>
using namespace std;
int main()
{
    ios::sync_with_stdio(false);
    map<double,int> wait;
    int n;//数字个数
    cin>>n;
    for(int count1=1;count1<=n;++count1)
    {
        double tem;
        cin>>tem;
        wait[tem]++;
    }
    for(auto c:wait)
    {
        for(int count1=1;count1<=c.second;++count1)
            cout<<c.first<<" ";
    }
}
```

(2)二维数组

用 map 实现二维数组很简单，类型定义为：map<int,map<int,int>> 如此就定义了一个 int 类型的二维数组。

(3)prim 算法

Prim 算法是来求最小生成树的经典算法，作为图论的一个算法，图的存储方式一般是用二维数组来储存的。在有了二维数组的前提情况下，就能够用 map 改写 prim 算法了。Prim 算法的大体思想并没有改变，只是在图的存储方式以及遍历方式加以改变。

除此以外，map 还有很多的方便之处，基于红黑树的 map 提供了像函数一项的映射关系，在编程中，就可以用 map 来存储变量之间的对应关系，而这一切都需要建立在红黑树的快速查找的基础之上。

```
map<int,map<int,int>> graph;
const int infinity=9999999;
int prim(int vertex)
{
    int sum=0;
    map<int,int> mincount;
    for(int count1=1;count1<=vertex;++count1)
        mincount[count1]=infinity;
    for(auto c:graph[1])
    {
        mincount[c.first]=c.second;
    }
    mincount[1]=0;
    for(int count1=1;count1<vertex;++count1)
    {
        int current,min1=infinity;
        for(auto c:mincount)
        {
            if(c.second!=0&&c.second<min1)
            {min1=c.second;current=c.first;}
        }
        if(min1==infinity)return 0;
        sum+=mincount[current];
        mincount[current]=0;
        for(auto c:graph[current])
        {
            if(mincount[c.first]&&c.second<mincount[c.first])
                mincount[c.first]=c.second;
        }
    }
    return sum;
}
```

4. 红黑树代码

```
#include<iostream>
#include<cstdlib>
#include<ctime>
using namespace std;
const int red=0;
const int black=1;
struct node
{
    int color=red;
```

```
    int number;
    node* left=NULL;
    node* right=NULL;
    node* parent=NULL;
};
node* root=NULL;
node* uncle(node* a)
{
    if(a==a->parent->left)return a->parent->right;
    else return a->parent->left;
}
int color(node *a)
{
    if(a==NULL)return black;
    else return a->color;
}
void leftrotate(node* x)
{
    node* right=x->right;
    x->right=right->left;
    if(right->left!=NULL)
        right->left->parent=x;
    right->parent=x->parent;
    if(x==root)
        root=right;
    else if(x==x->parent->left)
        x->parent->left=right;
    else
        x->parent->right=right;
    right->left=x;
    x->parent=right;
}
void rightrotate(node* x)
{
    node* left=x->left;
    x->left=left->right;
    if (left->right!=NULL)
        left->right->parent=x;
    left->parent=x->parent;
    if(x==root)
        root=left;
    else if(x==x->parent->right)
        x->parent->right=left;
    else
        x->parent->left=left;
    left->right=x;
```

```
    x->parent=left;
}
void rebalance(node* a)
{
    node* p;
    node* g;
    node* u;
    p=a->parent;
    while(color(p)==red&&p!=NULL&&a!=NULL)
    {
        g=p->parent;
        if(p==g->left)
        {
            u=g->right;
            if(color(u)==red)
            {
                u->color=black;
                p->color=black;
                g->color=red;
                a=g;
            }
            else
            {
                if(p->right==a)
                {
                    leftrotate(p);
                    node* tem=p;
                    p=a;
                    a=tem;
                }
                p->color=black;
                g->color=red;
                rightrotate(g);
            }
        }
        else
        {
            u=g->left;
            if(color(u)==red)
            {
                u->color=black;
                p->color=black;
                g->color=red;
                a=g;
            }
            else
```

```
            {
                if(p->left==a)
                {
                    rightrotate(p);
                    node* tem=p;
                    p=a;
                    a=tem;
                }
                p->color=black;
                g->color=red;
                leftrotate(g);
            }
        }
    }
    root->color=black;
    return;
}
void insert(int n)
{
    node* p=NULL;
    node* current=root;
    while(current!=NULL)
    {
        p=current;
        if(n>current->number)
            current=current->right;
        else if(n<current->number) current=current->left;
        else if(n==current->number)return;
    }
    auto c=new node;
    c->parent=p;
    c->left=NULL;
    c->right=NULL;
    c->color=red;
    c->number=n;
    if(p)
    {
        if(p->number>n)
            p->left=c;
        else
            p->right=c;
    }
    else root=c;
    rebalance(c);
}
void inorder(node* a)
```

```
{
    if(a==NULL)return;
    if(a->left!=NULL)inorder(a->left);
    cout<<a->number<<" ";
    if(a->right!=NULL)inorder(a->right);
}
int height(node* a,int heightnow)
{
    if(a->left==NULL&&a->right==NULL)return heightnow;
    int tem1=-1,tem2=-1;
    if(a->left!=NULL)
    {
        tem1=height(a->left,heightnow+1);
    }
    if(a->right!=NULL)
        tem2=height(a->right,heightnow+1);
    if(tem1>tem2)return tem1;
    else return tem2;
}
int main()
{
    srand(time(NULL));
    for(int count1=1;count1<=1000;++count1)
    {
        int tem=rand()%1000;
        cout<<tem<<" " << flush;
        insert(tem);
    }
    //cout<<height(root,0);
    //inorder(root);
}
```

参考文献：

【1】《算法》Robert Sedgewick,Kevin Wayne 北京:人民邮电出版社 2012年10月第一版
【2】《STL 源码剖析》侯捷 新竹·台湾 华中科技大学出版社 2015年12月