第十周讨论课资料

搜索树 Search trees

概述

> 红黑树（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Rudolf Bayer发明的，当时被称为平衡二叉B树（symmetric binary B-trees）。后来，在1978年被 Leo J. Guibas 和 Robert Sedgewick 修改为如今的“红黑树”。
> 红黑树和AVL树类似，都是在进行插入和删除操作时通过特定操作保持二叉查找树的平衡，从而获得较高的查找性能。
> 它虽然是复杂的，但它的最坏情况运行时间也是非常良好的，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。
> 数据

数据结构

> 红黑树，顾名思义，通过红黑两种颜色保证树的高度近似平衡。它的每个节点是一个五元组：color（颜色），key（数据），left（左孩子），right（右孩子）和p（父节点）。
> 红黑树的定义有以下五条：
>
> - 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）
> - 如果一个节点是红的，则它的两个儿子都是黑的
> - 从任一节点到其叶子的所有简单路径都包含相同数目的黑色节点。

关键操作实现

- 插入

> 1. 查找要插入的位置，从root节点向下走，直到找到一个叶子结点。时间复杂度为：O(N)
> 2. 将新节点的color赋为红色 这样利于后面调整
> 3. 自下而上重新调整该树为红黑树

```
#include<stdio.h>
#include<algorithm>
#include<vector>
#include<iostream>
using namespace std;
typedef struct rect {
    int id;
    int length;
    int width;
//对于向量元素是结构体的，可在结构体内部定义比较函数，下面按照id,length,width升序排序。
    bool operator<(const rect &a) const {
        if(id != a.id){
            return id < a.id;
        } else {
```

```
            if(length != a.length)
                return length < a.length;
            else
                return width < a.width;
        }
    }
} Rect;
int main() {
    vector<Rect> vec;
    Rect rect;
    rect.id=1;
    rect.length=2;
    rect.width=3;
    vec.push_back(rect);
    vector<Rect>::iterator it=vec.begin();
    cout<<(*it).id<<' '<<(*it).length<<' '<<(*it).width<<endl;
    return 0;
}
```

应用

> 红黑树和AVL树一样都对插入时间、删除时间和查找时间提供了最好可能的最坏情况担保。这不只是使它们在时间敏感的应用如即时应用(real time application)中有价值，而且使它们有在提供最坏情况担保的其他数据结构中作为建造板块的价值；例如，在计算几何中使用的很多数据结构都可以基于红黑树。
> 红黑树在函数式编程中也特别有用，在这里它们是最常用的持久数据结构之一，它们用来构造关联数组和集合，在突变之后它们能保持为以前的版本。除了O(log n)的时间之外，红黑树的持久版本对每次插入或删除需要O(log n)的空间。
> 红黑树是 2-3-4树的一种等同。换句话说，对于每个 2-3-4 树，都存在至少一个数据元素是同样次序的红黑树。在2-3-4 树上的插入和删除操作也等同于在红黑树中颜色翻转和旋转。这使得 2-3-4 树成为理解红黑树背后的逻辑的
> 重要工具，这也是很多介绍算法的教科书在红黑树之前介绍 2-3-4 树的原因，尽管 2-3-4 树在践中不经常使用。

Heaps——斐波那契堆

概述

斐波那契堆(Fibonacci heap)是计算机科学中树的集合。它比二项式堆具有更好的平摊分析性能，可用于实现合并优先队列。不涉及删除元素的操作有O(1)的平摊时间。 Extract-Min和Delete的数目和其它相比，较小时效率更佳。

数据结构

```
typedef int Type;
typedef struct _FibonacciNode {
    Type key; // 关键字(键值)
    int degree; // 度数
    struct _FibonacciNode *left; // 左兄弟
    struct _FibonacciNode *right; // 右兄弟
    struct _FibonacciNode *child; // 第一个孩子节点
    struct _FibonacciNode *parent; // 父节点
    int marked; //是否被删除第1个孩子(1表示删除，0表示未删除)
} FibonacciNode, FibNode;
```

FibNode是斐波那契堆的节点类，它包含的信息较多。key是用于比较节点大小的，degree是记录节点的度，left和right分别是指向节点的左右兄弟，child是节点的第一个孩子，parent是节点的父节点，marked是记录该节点是否被删除第1个孩子(marked在删除节点时有用)。

关键的基本操作的具体实现概述

- 插入

> 插入操作非常简单：插入一个节点到堆中，直接将该节点插入到"根链表的min节点"之前即可；若被插入节点比"min节点"小，则更新"min节点"为被插入节点。

- 合并

> 合并操作和插入操作的原理非常类似：将一个堆的根链表插入到另一个堆的根链表上即可。简单来说，就是将两个双链表拼接成一个双向链表。

Tries——HASH树

概述

> 在将一个数进行哈希的时候，我曾经写过关于哈希的这么些东西：对于数，当一个质数不够用的时候，可以加上第二个质数，用两个mod来确定该数据在库中的位置。那么这里需要简单的解释一下，对于一个质数x，它的mod有[0 .. x - 1] x种；所以对于两个质数x和y，能存储的无一重复的数据有 xy *个，其实也就是开一个*xy的二维数组。但是当数据极其多时，用两个质数去mod显然也是有不够的时候，就还要再加一个。为了便于查找，选取最小的十个质数，也就是2，3，5，7，11，13，17，23，29，31来mod，能包括的数就有10555815270个，已经远超出longint了。就是说，如果我开一个十维数组，那么取到一个数的效率就是O(1)！但是那样显然太浪费空间了，就可以用到树。
>
> 同一节点中的子节点，从左到右代表不同的余数结果。例如：第二层节点下有三个子节点。那么从左到右分别代表：除3余0，除3余1和除3余2。
>
> 对质数的余数决定了处理的路径。例如：某个数来到第二层子节点，那么它要做取余操作，然后再决定从哪个子节点向下搜寻。如果这个数是12那么它需要从第一个子节点向下搜索；如果这个数是7那么它需要从第二个子节点向下搜索；如果这个数是32那么它需要从第三个子节点向下搜索。这就是一个哈希树了。

数据结构

> 1. 结构简单
>
>   从哈希树的结构来说，非常的简单。每层节点的子节点个数为连续的质数。子节点可以随时创建。因此哈希树的结构是动态的，也不像某些哈希算法那样需要长时间的初始化过程。哈希树也没有必要为不存在的关键字提前分配空间 。
>
>   需要注意的是哈希树是一个单向增加的结构，即随着所需要存储的数据量增加而增大。即使数据量减少到原来的数量，但是哈希树的总节点数不会减少。这样做的目的是为了避免结构的调整带来的额外消耗。
>
> 2. 查找迅速
>
>   从算法过程我们可以看出，对于整数，哈希树层级最多能增加到10。因此最多只需要十次取余和比较操作，就可以知道这个对象是否存在。这个在算法逻辑上决定了哈希树的优越性。
>
>   一般的树状结构，往往随着层次和层次中节点数的增加而导致更多的比较操作。操作次数可以说无法准确确定上限。而哈希树的查找次数和元素个数没有关系。如果元素的连续关键字总个数在计算机的整数（32bit）所能表达的最大范围内，那么比较次数就最多不会超过10次，通常低于这个数值。

> 3. 结构不变
> 从删除算法中可以看出，哈希树在删除的时候，并不做任何结构调整。这个也是它的一个非常好的优点。常规树结构在增加元素和删除元素的时候都要做一定的结构调整，否则他们将可能退化为链表结构，而导致查找效率的降低。哈希树采取的是一种“见缝插针”的算法，从来不用担心退化的问题，也不必为优化结构而采取额外的操作，因此大大节约了操作时间。

插入

```
void insert(HashNodeentry, int level, Key key, Value value) {
    if (this.occupied = false) {
    this.key = key;
    this.value = value;
    this.occupied = true;
    return;
}
    int index = key
    mod Prime[level];
    if (nodes[index] = NULL) {
        nodes[index] = newHashNode();
    }
    level = level + 1;
    insert(nodes[index], level, key, value);
}
```

查找

```
Boolean search(HashNodeentry, int level, Key key, Value value) {
    if (this.occupied = true) {
        if (this.key = key) {
            value = this.value;
            return true;
        }
    }
    int index = key
    mod Prime[level];
    if (nodes[index] = NULL) {
        return false;
    }
    level = level + 1;
    return search(nodes[index], level, key, value);
}
```

应用

> 哈希树可以广泛应用于那些需要对大容量数据进行快速匹配操作的地方。例如：数据库索引系统、短信息中的收条匹配、大量号码路由匹配、信息过滤匹配。哈希树不需要额外的平衡和防止退化的操作，效率十分理想。

Spatial data partitioning trees——R树

概述

> R树是用于空间访问方法的树数据结构，即用于索引诸如地理坐标，矩形或多边形的多维信息。R-tree由AntoninGuttman于1984年提出，并且在理论和应用背景下都有重要用途。R树的常见实际用法可能是存储地理空间对象，
> 例如餐馆位置或典型地图由以下构成的多边形：街道，建筑物，湖泊轮廓，海岸线等，然后快速查找答案的答案例如“查找距离我当前位置2公里范围内的所有博物馆”，“检索我所在位置2公里范围内的所有路段”对于各种距离度量，R树还可以加速最近邻搜索，包括大圆距离。

数据结构

> R树是B树在高维空间的扩展，是一棵平衡树。每个R树的叶子结点包含了多个指向不同数据的指针，这些数据可以是存放在硬盘中的，也可以是存在内存中。根据R树的这种数据结构，当我们需要进行一个高维空间查询时，我们只需要遍历少数几个叶子结点所包含的指针，查看这些指针指向的数据是否满足要求即可

一棵R树满足如下的性质:

1. 除根结点之外，所有非根结点包含有m至M个记录索引(条目)。根结点的记录个数可以少于m。通常，m=M/2。
2. 对于所有叶子中存储的记录(条目)，I是最小的可以在空间中完全覆盖这些记录所代表的点的矩形(注意:此处所说的”矩形”是可以扩展到高维空间的)。
3. 对于所有非叶子结点上的记录(条目)，i是最小的可以在空间上完全覆盖这些条目所代表的点的矩形(同性质2)。
4. 所有叶子结点都位于同一层，因此R树为平衡树。

参考资料

[1] 百度百科. (无日期). 红黑树. 检索来源: 百度:https://baike.baidu.com/item/%E7%BA%A2%E9%BB%91%E6%A0%91/2413209?fr=aladdin
[2] 博客园. (无日期). 斐波那契堆(一)之 图文解析 和 C语言的实现. 检索来源: 百度: https://www.cnblogs.com/skywang12345/p/3659060.html
[3] CSDN. (无日期).痴人说Hash - 哈希树 (HashTree). 检索来源: 百度:https://blog.csdn.net/chinaleesunnyboy/article/details/79689542