




树形结构

内容简要

- ❑ Search Tree (搜索树)
- ❑ Tries (字典树)
- ❑ Heaps(堆树)
- ❑ Spatial data partitioning trees
- ❑ 每部分介绍包括：概述，数据结构的定义，关键基本操作具体实现的概述(1-2),应用(求解具体问题)



搜索树(Search Tree)

- 概述
- Treap的定义
- 关键基本操作具体实现的概述
- 应用(求解具体问题)



概述

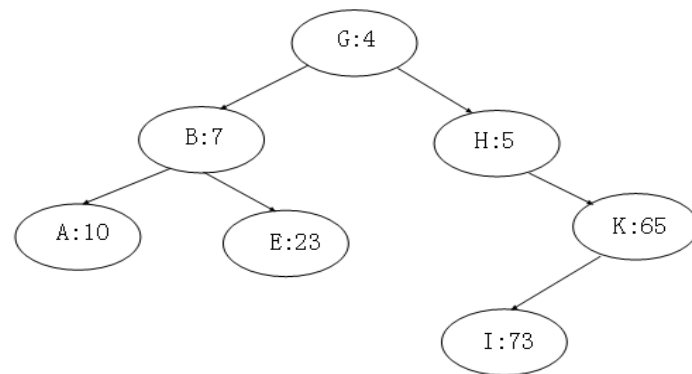
搜索树包含二叉排序树，平衡二叉树，红黑树，伸展树，Treap等多种树形结构，以Treap为代表介绍一下搜索树的主要特点

Treap又叫树堆，Treap=tree+heap,它是有一个随机附加域满足堆的性质的二叉搜索树,它的左子树和右子树也分别是一个Treap,和一般的二叉搜索树不同的是，Treap记录的是一个额外的数据，就是优先级。Treap在以关键码构成二叉搜索树的同时，还满足了堆的性质。



Treap的数据结构(定义)

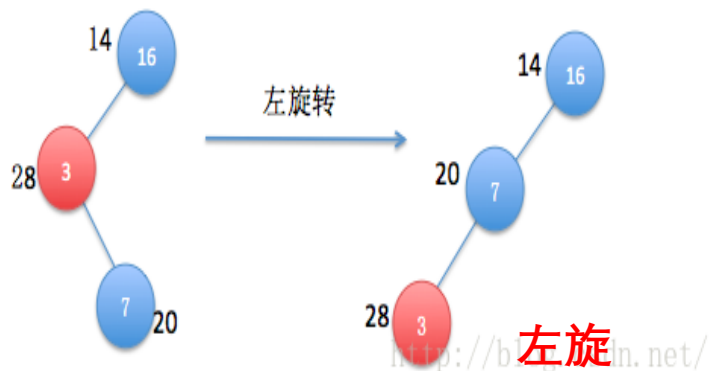
Treap每个节点需要记录两个数据，一个是键值，一个是随机附加的优先级，Treap在以关键码构成二叉排序树的同时，又以结点优先级形成最大堆和最小堆。所以Treap必须满足这两个性质，一是二叉排序树的性质，二是堆的性质





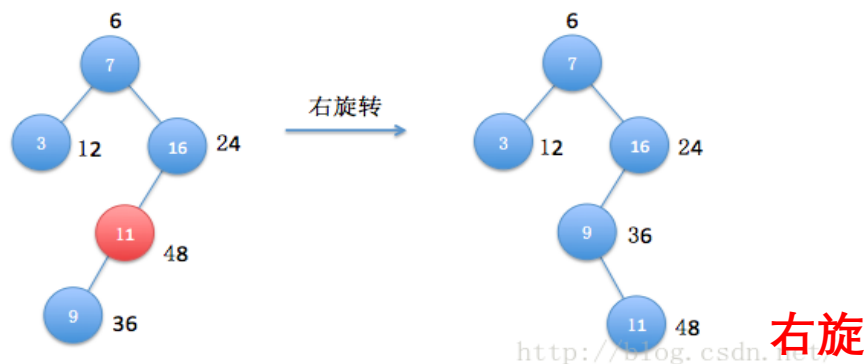
关键基本操作具体实现的基本概述

基本概念：旋转
(左旋，右旋)



//左旋转

```
void left_rotate(Tree &node)
{
    Tree temp=node->rchild;
    node->rchild=temp->lchild;
    temp->lchild=node;
    node=temp;
}
```



//右旋转

```
void right_rotate(Tree &node)
{
    Tree temp=node->lchild;
    node->lchild=temp->rchild;
    temp->rchild=node;
    node=temp;
}
```




关键基本操作具体实现的基本概述

插入

插入操作说明：给节点随机分配一个优先级，先把要插入的点插入到一个叶子上，然后再和维护堆的性质。

```
bool insert_val(Tree &root, Tree &node)
{
    if (!root)
    {
        root=node; //插入
        return true;
    }
    else if(root->val>node->val)
    {
        bool flag=insert_val(root->lchild, node);
        if (root->priority>node->priority) //检查是否需要调整
            right_rotate(root); //右旋
        return flag;
    }
    else if(root->val<node->val)
    {
        bool flag=insert_val(root->rchild, node);
        if (root->priority>node->priority) //检查是否需要调整
            left_rotate(root); //左旋
        return flag;
    }
    //已经含有该元素，释放结点
    delete node;
    return false;
}
```





关键基本操作具体实现的基本概述

删除

```
//删除函数
bool remove(Tree &root, ElementType val)
{
    if (root==NULL)
        return false;
    else if (root->val>val)
        return remove(root->lchild, val);
    else if (root->val<val)
        return remove(root->rchild, val);
    else //找到执行删除处理
    {
        Tree *node=&root;
        while ((*node)->lchild && (*node)->rchild) //从该结点开始往下调整
        {
            if ((*node)->lchild->priority<(*node)->rchild->priority) //比较其左右孩子优先
            {
                right_rotate(*node); //右旋转
                node=&(*node)->rchild; //更新传入参数, 进入下一层
            }
            else
            {
                left_rotate(*node); //左旋转
                node=&(*node)->lchild; //更新传入参数, 进入下一层
            }
        }
    }
}
```

//最后调整到(或者本来是)叶子结点, 或者只有一个孩子的情况, 可以直接删除了

```
if ((*node)->lchild==NULL)
    (*node)=(*node)->rchild;
else if ((*node)->rchild==NULL)
    (*node)=(*node)->lchild;
return true;
```





应用(相关操作举例和求解具体问题)

题目

Description

有一个序列含有一定数量的元素，现在要求写一个程序，满足以下几个要求：

- 【A】支持插入操作（这个序列不允许有重复元素，即是说，如果待插入的元素已经出现在这个序列中，那么请忽略此操作）
- 【B】支持删除操作（如果此序列中不包含待删除的这个元素，则忽略此操作，否则删除这个元素）
- 【C】查找元素 x 的前继元素（前继元素是指：小于 x 且与 x 最接近的元素，当然，如果 x 已经是序列中的最小元素，则 x 没有前继元素）
- 【D】查找元素 x 的后继元素（后继元素是指：大于 x 且与 x 最接近的元素，当然，如果 x 已经是序列中的最大元素，则 x 没有后继元素）
- 【E】找第 k 小的元素
- 【F】求某个元素 x 的秩（即 x 的排名是多少，从小到大排序）



应用(相关操作举例和求解具体问题)

输入输出要求

Input

多组数据（整个文件以输入 -1 结束）

对于每组数据，有若干行（最多100000行），表示的意义如下：

【A】 insert x

【B】 delete x

【C】 predecessor x

【D】 successor x

【E】 Kth x

【F】 rank x

这6种操作的意义与上面的定义相对应！

【G】 print 表示从小到大输出序列中的所有元素

【H】 end 表示结束本组数据

每组输入数据后有一空行！

Output

对于以上8种操作，分别输出对应信息，如下：

【A】 insert x 不用输出任何信息

【B】 delete x 如果x存在，则删除x，否则输出 Input Error

【C】 predecessor x 如果x不存在，输出 Input Error；否则如果x是序列中的最小元素，输出对应信息（见样例），否则输出x的前继元素

【D】 successor x 如果x不存在，输出 Input Error；否则如果x是序列中的最大元素，输出对应信息（见样例），否则输出x的后继元素

【E】 Kth x 如果x不合法，输出 Input Error；否则输出第Kth小的元素（见样例）

【F】 rank x 如果x不存在，输出 Input Error；否则输出x的排名（见样例）

【G】 print 从小到大输出序列中的所有元素，每个元素后加一个逗号，并在最后加上 end of print（见样例）

【H】 end 输出 end of this test



应用(相关操作举例和求解具体问题)

利用Treap树的特点，可以很好地实现插入，删除，排名等操作，首先需要写Treap相关树的基本操作函数，在做这个题目的时候在进行相关调用

```
node *find(node*o, int x)//查找
{
    if (o == nullptr) return 0;
    if (o->v == x) return o;
    else if (o->v > x)
        return find(o->ch[0], x);
    else return find(o->ch[1], x);
}
void clear(node *&o)//清空
{
    if (o == nullptr) return;
    if (o->ch[0] != nullptr) clear(o->ch[0]);
    if (o->ch[1] != nullptr) clear(o->ch[1]);
    o = nullptr;
}
void print(node *o)//中序遍历
{
    if (o)
    {
        print(o->ch[0]);
        printf("%d,", o->v);
        print(o->ch[1]);
    }
}
```

样例：输入输出

```
insert 20
insert 5
insert 1
insert 15
insert 9
insert 25
insert 23
insert 30
insert 35
print
Kth 0
Kth 1
Kth 3
Kth 5
Kth 7
Kth 9
Kth 10
rank 1
rank 3
rank 5
rank 15
rank 20
rank 30
rank 31
rank 35
successor 15
successor 35
successor 25
successor 26
predecessor 1
predecessor 20
predecessor 23
predecessor 15
predecessor 111
delete 9
delete 15
delete 25
delete 23
delete 20
print
Kth 3
Kth 4
rank 30
rank 35
end
```

```
1,5,9,15,20,23,25,30,35,end of print
Input Error
The 1_th element is 1
The 3_th element is 9
The 5_th element is 20
The 7_th element is 25
The 9_th element is 35
Input Error
The rank of 1 is 1_th
Input Error
The rank of 5 is 2_th
The rank of 15 is 4_th
The rank of 20 is 5_th
The rank of 30 is 8_th
Input Error
The rank of 35 is 9_th
The successor of 15 is 20
35 is the maximum
The successor of 25 is 30
Input Error
1 is the minimum
The predecessor of 20 is 15
The predecessor of 23 is 20
The predecessor of 15 is 9
Input Error
1,5,30,35,end of print
The 3_th element is 30
The 4_th element is 35
The rank of 30 is 3_th
The rank of 35 is 4_th
end of this test
```

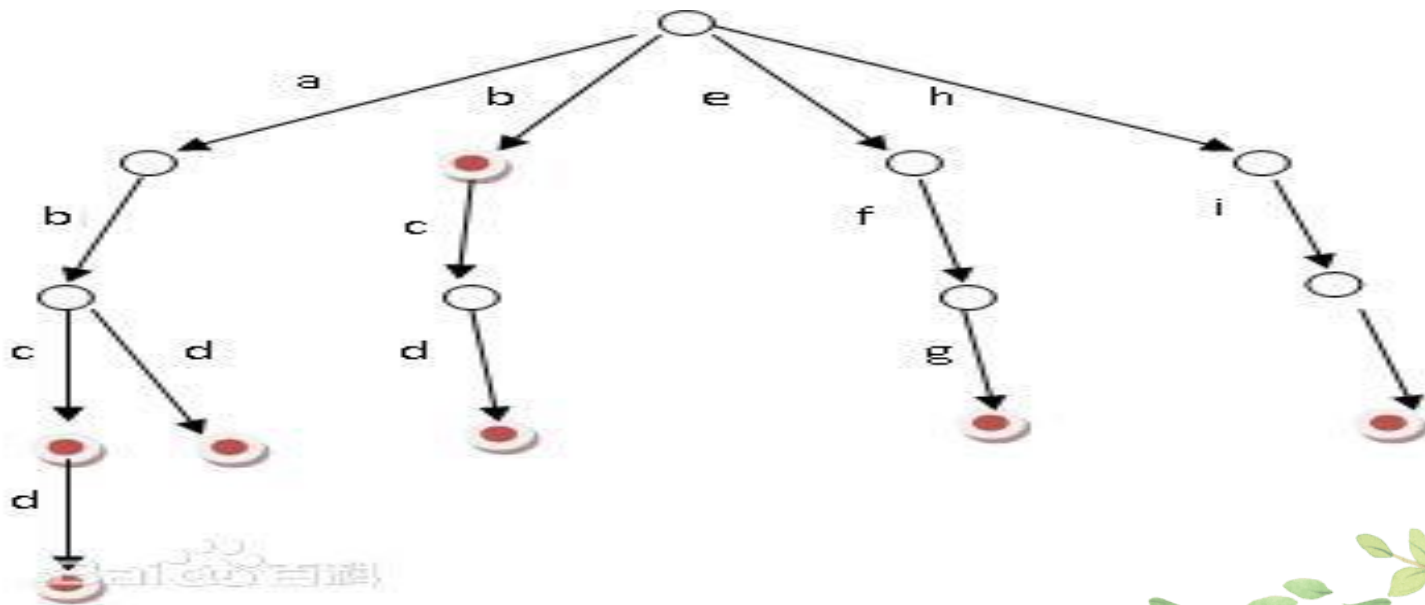


Tries (字典树)

- 概述
- Trie的定义
- 关键基本操作具体实现的概述
- 应用(求解具体问题)



字典树又称单词查找树，Trie树，是一种树形结构，是一种哈希树的变种。典型应用是用于统计，排序和保存大量的字符串(但不仅限于字符串)，所以经常被搜索引擎系统用于文本词频的统计。它的优点是：利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较，查询效率比哈希树高。





字典树的数据结构(定义)



1

- 根节点不包含字符,除根节点外每一个节点都只包含一个字符
- 从根节点到某一节点,路径上经过的字符连接起来,为该节点对应的字符串
- 每个节点的所有子节点包含的字符都不相同。

2

- 通常在实现的时候,会在结点结构中设置一个标志,用来标记该节点处是否构成一个单词 (关键字: count)。
- Trie树的关键字一般都是字符串,而且Trie树把每个关键字保存在一条路径上,而不是一个节点中。



关键步骤的实现和操作

搜索字典项目的方法为:

- (1)从根结点开始一次搜索;
- (2)取得要查找关键词的第一个字母,并根据该字母选择对应的子树并转到该子树继续进行检索;
- (3)在相应的子树上,取得要查找关键词的第二个字母,并进一步选择对应的子树进行检索。
- (4)迭代过程……
- (5)在某个结点处,关键词的所有字母已被取出,则读取附在该结点上的信息,即完成查找。

其他操作类似处理

字典树的插入操作

```
void trie_insert(trie root, char* key)
{
    trie_node* node = root;
    char* p = key;
    while(*p)
    {
        if(node->children[*p-'a'] == NULL)
        {
            node->children[*p-'a'] = create_trie_node(); //创建树
        }
        node = node->children[*p-'a'];
        ++p;
    }
    node->count += 1;
}
```

说明:

count是记录该节点代表的单词的个数
children是存储各个子结点
create_trie_node()是创建字典树的新结点



关键步骤的实现和操作

查询操作：找到返回1，找不到就返回0

```
int trie_search(trie root, char* key)
{
    trie_node* node = root;
    char* p = key;
    while(*p && node!=NULL)
    {
        node = node->children[*p-'a'];
        ++p;
    }
    if(node == NULL)
        return 0;
    else
        return node->count;
}
```

说明：
count是记录该节点代表的单词的个数
children是存储各个子结点
create_trie_node()是创建字典树的新结点



对于Trie树，一般只需实现查询和插入操作，就可以对实现单词的检索和词频统计的功能





应用(相关操作举例和求解具体问题)

简单应用举例

- 串的快速检索

给出N个单词组成的熟词表,以及一篇全用小写英文书写的文章,请你按最早出现的顺序写出所有不在熟词表中的生词。

在这道题中,我们可以用数组枚举,用哈希,用字典树,先把熟词建一棵树,然后读入文章进行比较,这种方法效率是比较高的。

- “串”排序

给定N个互不相同的仅由一个单词构成的英文名,让你将他们按字典序从小到大输出

用字典树进行排序,采用数组的方式创建字典树,这棵树的每个结点的所有儿子很显然地按照其字母大小排序。对这棵树进行先序遍历即可

- 最长公共前缀

对所有串建立字典树,对于两个串的最长公共前缀的长度即他们所在的结点的公共祖先个数,于是,问题就转化为当时公共祖先问题。



应用(相关操作举例和求解具体问题)

例子：编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀，返回空字符串 ""


算法思想

利用字典树解题：以其中一个字符串为标准，遍历该字符串在树中的节点信息，第一个出现分支的节点即为要找的分割点，从根节点到该点的字符串即为所求最长公共前缀

主要代码实现



```
string longestCommonPrefix(vector<string>& strs)
{
    int n = strs.size();
    if (n == 0)
    {
        return ""; //当输入数组为空时，返回空字符串
    }
    for (int i = 0; i < n; i++)
    {
        if (strs[i].size() == 0) //数组中某一字符串为空时，
            则返回空字符串
            return "";
        insert(strs[i]); //前缀树的特点：在查询前先插入字符串
    }
    string result = startsWith(strs[0], n); //查询过程
    return result;
} //startsWith(),size()为字典树的相关操作
```

堆 (heap)

- 概述
- heap的定义
- 关键基本操作具体实现的概述
- 应用(求解具体问题)



概述

- ❑ 堆(Heap)是计算机科学中一类特殊的数据结构的统称。堆通常是一个可以被看做一棵完全二叉树的数组对象。
- ❑ 堆的定义如下： n 个元素的序列 $\{k_1, k_2, k_i, \dots, k_n\}$ 当且仅当满足下关系时，称之为堆。
($k_i \leq k_{2i}, k_i \leq k_{2i+1}$)或者($k_i \geq k_{2i}, k_i \geq k_{2i+1}$), ($i = 1, 2, 3, 4 \dots n/2$)
- ❑ 将根节点最大的堆叫做最大堆或大根堆，根节点最小的堆叫做最小堆或小根堆。常见的堆有二叉堆、斐波那契堆。以Fibonacci heap为代表介绍一下堆的特点



Fibonacci heap的数据结构(定义)

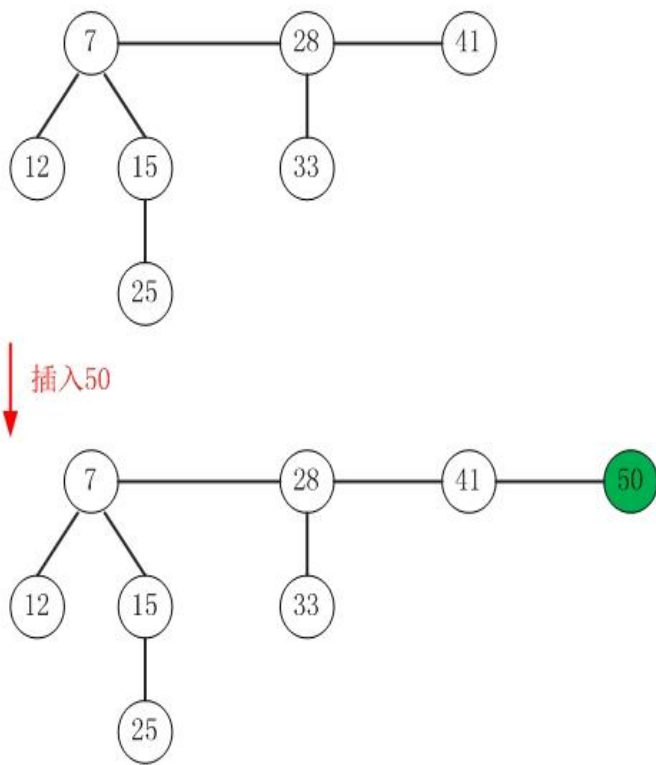
- ❑ 斐波那契堆是由一组最小堆有序树构成的。每个节点的度数为其子节点的数目。树的度数为其根节点的度数
- ❑ 斐波那契堆中的树都是有根的但是无序。每个节点 x 包含指向父节点的指针 $p[x]$ 和指向任意一个子结点的 $child[x]$ 。 x 的所有子节点都用双向循环链表链接起来，叫做 x 的子链表。子链表中的每一个节点 y 都有指向它的左兄弟的 $left[y]$ 和右兄弟的 $right[y]$ 。如果节点 y 是 x 仅有的子节点，则 $left[y]=right[y]=y$ 。
- ❑ 斐波那契堆中所有树的根节点也用一个双向循环链表链接起来。
- ❑ 使用一个指针指向斐波那契堆中最小元素



关键基本操作具体实现的基本概述

插入操作示意图及代码

添加“节点50”



//注意此处node是单个节点，而root是双向链表

```
static void fib_node_add(FibNode *node, FibNode *root)
```

```
{
```

```
    node->left=root->left;
```

```
    root->left->right=node;
```

```
    node->right=root;
```

```
    root->left=node;
```

```
}
```

//将节点node插入到斐波那契堆heap中

```
static void fib_heap_insert_node(FibHeap *heap, FibNode *node)
```

```
{
```

```
    if (heap->keyNum == 0)
```

```
        heap->min = node;
```

```
    else
```

```
    {
```

```
        fib_node_add(node, heap->min);
```

```
        if (node->key < heap->min->key)
```

```
            heap->min = node;
```

```
    }
```

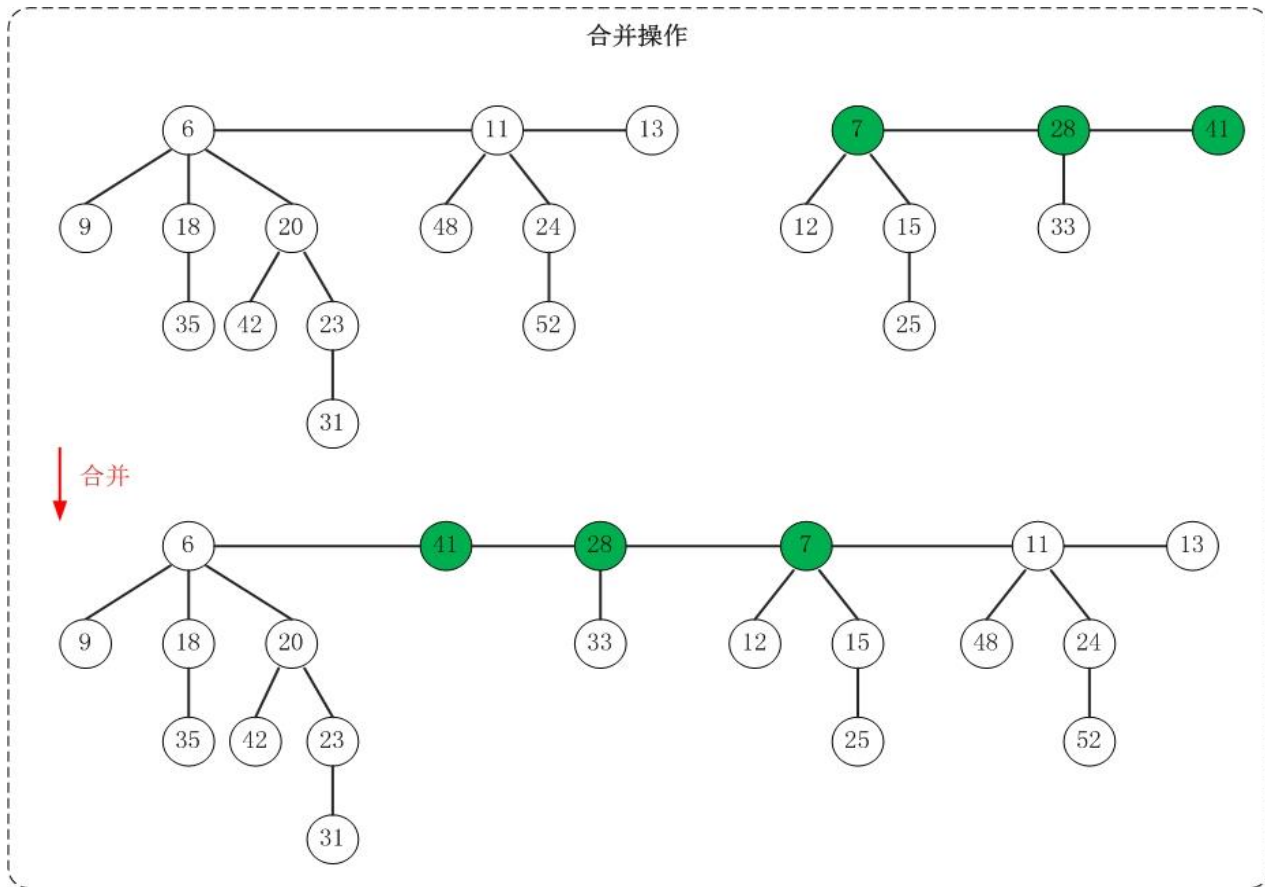
```
    heap->keyNum++;
```

```
}
```



关键基本操作具体实现的基本概述

合并操作示意图及代码(一)





关键基本操作具体实现的基本概述

将双向链表b链接到双向链表a的后面

注意：此处a和b都是双向链表

*/

```
static void fib_node_cat(FibNode *a, FibNode *b)
```

```
{
```

```
    FibNode *tmp;  
    tmp=a->right;  
    a->right=tmp;  
    b->right->left=a;  
    b->right=tmp;  
    tmp->left=b;
```

```
}
```

//将h1, h2合并成一个堆, 并返回合并后的堆

```
FibHeap* fib_heap_union(FibHeap *h1, FibHeap *h2)
```

```
{
```

```
    FibHeap *tmp;
```

```
    if (h1==NULL)  
        return h2;  
    if (h2==NULL)  
        return h1;
```

// 以h1为“母”，将h2附加到h1上；下面是保证h1的度数大，尽可能的少操作。

```
    if(h2->maxDegree > h1->maxDegree)
```

```
    {
```

```
        tmp = h1;  
        h1 = h2;  
        h2 = tmp;
```

```
    }
```

合并操作示意图及代码(二)

```
if((h1->min) == NULL)
```

// h1无“最小节点”

```
{
```

```
    h1->min = h2->min;  
    h1->keyNum = h2->keyNum;  
    free(h2->cons);  
    free(h2);
```

```
}
```

```
else if((h2->min) == NULL)
```

// h1有“最小节点” && h2无“最小节点”

```
{
```

```
    free(h2->cons);  
    free(h2);
```

```
}
```

// h1有“最小节点” && h2有“最小节点”

```
else
```

```
{
```

```
    // 将“h2中根链表”添加到“h1”中  
    fib_node_cat(h1->min, h2->min);  
    if (h1->min->key > h2->min->key)  
        h1->min = h2->min;  
    h1->keyNum += h2->keyNum;  
    free(h2->cons);  
    free(h2);
```

```
}
```

```
return h1;
```

```
}
```



应用(相关操作举例和求解具体问题)

- ❑ 斐波那契堆 (Fibonacci heap) 是堆中一种，它和二项堆一样，也是一种可合并堆，可用于实现合并优先队列。
- ❑ 斐波那契堆比二项堆具有更好的平摊分析性能，它的合并操作的时间复杂度是 $O(1)$ ，可以用来优化空间。



A stylized, colorful illustration of a landscape. On the left, there are two large trees with brown trunks and green and yellow foliage. In the background, there are blue and white wavy lines representing hills or clouds. A small yellow sun with a face is in the top left corner. A pink speech bubble with a dashed border contains the text 'Sptatial-date partitioning trees'.

Sptatial-date partitioning trees

- 概述
- Sptatial date partitioning trees 的定义
- 关键基本操作具体实现的概述
- 应用(求解具体问题)



概述

- Spatial data partitioning trees 名为统计数据划分树，其典型代表有Burkhard-Keller树，一种基于树的数据结构，曾解决了Levenshtein距离问题

- BK树或者称为Burkhard-Keller树，是一种基于树的数据结构，被设计于快速查找近似字符串匹配，比方说拼写检查器，或模糊查找，当搜索"aek"时能返回"seek"和"peek"。
- 为何BK-Trees这么酷，因为除了穷举搜索，没有其他显而易见的解决方法，并且它能以简单和优雅的方法大幅度提升搜索速度。

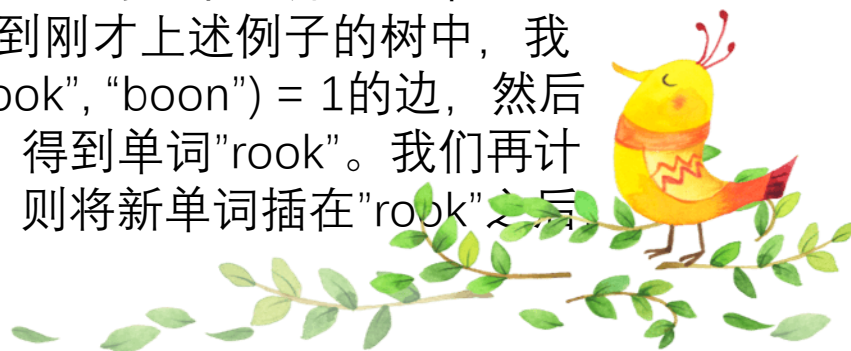




BK树的数据结构(定义)

```
struct node
{
    char word[30]; //当前结点值
    node *next[30];
} root; //结构体类型的定义
```

- BK树类似于一棵字典树，通过构造一棵BK树，可以从构造过程中去体会到BK树具体的结构特点
- 每个节点有任意个子节点，每条边有个值表示编辑距离。所有子节点到父节点的边上标注n表示编辑距离恰好为n。比如，我们有棵树父节点是"book"和两个子节点"rook"和"nooks"，"book"到"rook"的边标号1，"book"到"nooks"的边上标号2。
- 从字典里构造好树后，无论何时你想插入新单词时，计算该单词与根节点的编辑距离，并且查找数值为 $d(\text{newword}, \text{root})$ 的边。递归得与各子节点进行比较，直到没有子节点，你就可以创建新的子节点并将新单词保存在那。比如，插入"boon"到刚才上述例子的树中，我们先检查根节点，查找 $d(\text{"book"}, \text{"boon"}) = 1$ 的边，然后检查标号为1的边的子节点，得到单词"rook"。我们再计算距离 $d(\text{"rook"}, \text{"boon"}) = 2$ ，则将新单词插在"rook"之后，边标号为2。





关键步骤的实现和操作

```
//建树
void insert(node*q, char*str)
{
    node*l=q;
    while(l)
    {
        int dis=diff(l->word, str);
        if(!l->next[dis])
        {
            l->next[dis]=&p[num++];
            strcpy(l->next[dis]->word+1, str+1);
            break;
        }
        l=l->next[dis];
    }
}
```

说明



构造一棵BK树，diff函数是用来进行动态规划验算的，内调用函数init()对dp[][]进行初始化



调用insert函数可以构造一棵完整的BK树型结构



关键步骤的实现和操作

查找操作

```
//查找与单词相差不大于d的单词???  
void sfind(node*q, char*str, int d)  
{  
    if(flag)  
        return;  
    node*l=q;  
    if(l==NULL)  
        return;  
    int dis=diff(str, l->word);  
    if(dis<=d)  
    {  
        fuck++;  
    }  
    for(int x=dis-d; x<=dis+d; x++)  
    {  
        if(x>=0&& x<=20&& l->next[x])  
            sfind(l->next[x], str, d);  
    }  
}
```





应用(相关操作举例和求解具体问题)

解决编辑距离问题

字符串A到B的编辑距离是指, 只用插入、删除和替换三种操作, 最少需要多少步可以把A变成B



1. 首先观察Levenshtein距离的性质。令 $d(x,y)$ 表示字符串x到y的Levenshtein距离, 那么显然: 1. $d(x,y) = 0$ 当且仅当 $x=y$ (Levenshtein距离为0 \iff 字符串相等) 2. $d(x,y) = d(y,x)$ (从x变到y的最少步数就是从y变到x的最少步数) 3. $d(x,y) + d(y,z) \geq d(x,z)$ (从x变到z所需的步数不会超过x先变成y再变成z的步数)
2. 最后这一个个性质叫做三角形不等式。就好像一个三角形一样, 两边之和必然大于第三边。给某个集合内的元素定义一个二元的“距离函数”, 如果这个距离函数同时满足上面说的三个性质, 我们就称它为“度量空间”。



应用(相关操作举例和求解具体问题)

例子：求算两个字符串之间的编辑距离
简述

设A和B是两个字符串，要用最少的字符操作将字符串A转换为字符串B

字符串操作包括，

- 1) 删除一个字符
- 2) 插入一个字符
- 3) 将一个字符改为另一个字符

算法说明

模拟构造一个 $(m + 1)$ 行， $(n + 1)$ 列的表格每一次都是在前一次的计算结果下，得到当前的值首先是三个特殊情况 用srcStr表示源字符串，dstStr 表示目标字符串

1) 两个空字符串的编辑距离 $D(\text{srcStr}, \text{dstStr}) = 0$ 2) 如果srcStr为空，dstStr不为空，则 $D(\text{srcStr}, \text{dstStr}) = \text{dstStr.length}()$ ，即在原空字符串上添加字符，形成dstStr 3) 如果dstStr为空，srcStr不为空，则 $D(\text{srcStr}, \text{dstStr}) = \text{srcStr.length}()$ ，及在源字符串上删除其所有字符，直至为空



THANKYOU!

参考文献：

- 【1】 胡凡，曾磊 。算法笔记【M】。北京：机械工业出版社，2016： 359-367.
- 【2】 百度百科词条
- 【3】 CDSN博客