**可合并堆的一种实现（转）**

http://blog.csdn.net/theprinceofelf/article/details/7247454

左偏树，也可以称之为左式堆。称其为树，是因为其存储结构通常采用二叉树，所以可以认为是一种特殊的二叉树。称其为堆，是因为在逻辑结构上，它属于可合并堆的一种。其实数据结构中最欣欣向荣的两个分支就是：平衡树 和可合并堆。高级树结构的核心都是围绕如何使树到达平衡而展开，高级堆结构的核心就是如何有效地进行合并。

这里先介绍可合并堆中一种十分好用的数据结构：左式堆。人们熟悉或者常听说的堆主要有：（二叉）堆、左式堆、斜堆、二项堆、斐波纳契堆。其中二叉堆是最为常用的，一般的优先队列都可以考虑用二叉堆实现，而STL中优先队列，更是方便了大家的使用，如果自己编程主要注意sift down 和 sift up 这两个核心操作的实现。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 二叉堆 | 左式堆 | 斜堆 | 二项堆 | 斐波纳契堆 |
| Insert | O(log n) | O(log n) | O(log n) | O(log n) | O(1) |
| Extract-Min | O(log n) | O(log n) | O(log n) | O(log n) | O(log n) |
| merge | O(m + n) | O(log n) | O(log n) | O(log n) | O(1) |
| 编程复杂度 | 简单 | 较简单 | 较简单 | 较复杂 | 复杂 |

左式堆由于使用二叉链表的存储结构，对动态操作支持良好，在需要合并操作的场合，是极佳的选择。二项堆（使用孩子兄弟存储模式）和斐波纳契堆（使用树型结构配合双向循环链表模式）由于编程的复杂性和比较大的常数因子，所以使用的很少。

下面进入主题leftist tree:

首先看左偏树的性质：

1.【堆性质】：任意节点的关键字大于等于其孩子节点的关键字

2.【左偏性质】：定义到最近的孩子的距离为节点距离dist，那么任意节点的左孩子的距离大于右孩子的距离。

A->lchild->dist >= A->rchild->dist

堆性质是为了让最小的结点始终在根的位置，这是所有堆都有的性质。

而左偏性质，则是为了让树状存储的堆，树的深度不能过大，且利于合并。

那么这个性质是怎么完成这两个功能的呢？左偏性质使树的左侧的深度始终大于等于右侧的深度，这一点从名字上就能体会到，也可以画几棵左偏的树试试。而左偏树在实现插入操作时总是从右侧插入，也就是总是让短的一侧生长，如果右侧长于了左侧，那么把左右侧交换一下，继续从短的一侧生长。其实如果不考虑具体的细节，那么这样的直观理解可以看到左偏树的一些本质内涵，一颗树有两个分支，每次要生长的时候，总是让短的一侧先生长，那么这棵树，最后是不是就能够比较对称呢？自然常识和严密的技术逻辑有时候是一致的。

下面给出左偏树的具体实现（做过简单测试，但不保证没有BUG。如果发现BUG请告知，我会及时修改，谢谢）：

[cpp] view plaincopy

struct LTNode

{

int data;

int dist;

struct LTNode \*lchild;

struct LTNode \*rchild;

};

template<typename T>

inline void swap(T &a,T &b)

{

T tmp = a;

a = b;

b = tmp;

}

void InOrder(LTNode \*t)

{

if(t)

{

InOrder(t->lchild);

printf("%d\n",t->data);

InOrder(t->rchild);

}

}

LTNode\* merge(LTNode\* &A,LTNode\* &B)

{

if(A==NULL || B==NULL)

return A==NULL?B:A;

if(A->data > B->data)

{ swap(A,B); }

A->rchild = merge(A->rchild,B);

if(A->lchild==NULL ||

A->rchild->dist > A->lchild->dist)

swap(A->lchild,A->rchild);

if(A->rchild==NULL){A->dist = 0;}

else {A->dist = A->rchild->dist + 1;}

return A;

}

void insert(LTNode \*&t,int x)

{

LTNode \*p = new LTNode;

p->data = x;

p->dist = 0;

p->lchild = NULL;

p->rchild = NULL;

t = merge(t,p);

printf("insert %d success\n",x);

}

LTNode\* ExtractMin(LTNode\* &t)

{

LTNode \*p = t;

t = merge(t->lchild,t->rchild);

return p;

}