斐波那契堆

文章分类:C++编程

学习的地方:

http://en.wikipedia.org/wiki/Talk:Fibonacci\_heap

http://jicheng.ycool.com/post.2380567.html

1,先贴个代码:

Cpp代码

#include <iostream>

using namespace std;

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

struct element

{

int key;

};

struct fiboheap

{

element data;

fiboheap \*child;

fiboheap \*left\_link;

fiboheap \*right\_link;

int degree;

bool child\_cut;

fiboheap \*parent;

};

#define TREELEN 8

void join\_min\_trees(fiboheap \*a, fiboheap \*b)

{

if(a->data.key < b->data.key)

{

if(a->child == NULL)

{

a->right\_link = a->left\_link = a;

b->right\_link = b->left\_link = b;

}

else

{

fiboheap \*temp = a->child->left\_link;

a->child->left\_link = b;

b->right\_link = a->child;

b->left\_link = temp;

temp->right\_link = b;

}

a->child = b;

a->degree++;

b->parent = a;

b->child\_cut = true;

}

else

{

if(b->child == NULL)

{

a->right\_link = a->left\_link = a;

b->right\_link = b->left\_link = b;

}

else

{

fiboheap \*temp = b->child->left\_link;

b->child->left\_link = a;

a->right\_link = b->child;

a->left\_link = temp;

temp->right\_link = a;

}

b->child = a;

b->degree++;

a->parent = b;

a->child\_cut = true;

a = b;

}

}

//寻找于所给值相同的节点

fiboheap \*find(fiboheap \*a, int key)

{

fiboheap \*p = a;

do {

if(p->data.key == key)

{

return p;

}

if (p->child != NULL)

{

fiboheap \*q = find(p->child, key);

if(q != NULL) return q;

}

p = p->right\_link;

} while(p != a);

return NULL;

}

element del\_min(fiboheap\*&);

//删除与所给值相同的节点

void del\_fix(fiboheap \*&a, int key)

{

fiboheap \*p = find(a, key);

if(p == NULL)

{//堆中没有与所给值相同的节点

return;

}

else if(p == a)

{//正好要删除最小值

del\_min(p);

} else if(p->child\_cut == false) {//删除处于顶点节点所组成的双向循环链表中

fiboheap \*i = p->left\_link;

fiboheap \*j = p->right\_link;

fiboheap \*k = p->child;

fiboheap \*m = k->left\_link;

i->right\_link = k;

k->left\_link = i;

m->right\_link = j;

j->left\_link = m;

k->parent = NULL;

k->child\_cut = false;

free(p);

}

else//删除不在由顶点节点所组成的双向循环链表中

{

if(p->degree == 0)//节点下面没有子节点了

{

p->parent->child = NULL;

p->parent->degree--;

} else//节点下面还有子节点

{

fiboheap \*p\_left = p->left\_link;

fiboheap \*p\_right = p->right\_link;

p\_left->right\_link = p\_right;

p\_right->left\_link = p\_left;

p->parent->child = p\_right;

p->parent->degree--;

fiboheap \*child = p->child;

child->parent = NULL;

child->child\_cut = false;

fiboheap \*child\_left = child->left\_link;

fiboheap \*a\_right = a->right\_link;

a->right\_link = child;

child->left\_link = a;

child\_left->right\_link = a\_right;

a\_right->left\_link = child\_left;

}

//级联剪枝操作

fiboheap \*del\_node = p;

p = p->parent;

while(p->child\_cut != false) {

fiboheap \*parent = p->parent;

p->parent = NULL;

fiboheap \*p\_left = p->left\_link;

fiboheap \*p\_right = p->right\_link;

p\_left->right\_link = p\_right;

p\_right->left\_link = p\_left;

p->child\_cut = false;

parent->child = p\_right;

parent->degree--;

fiboheap \*a\_right = a->right\_link;

a->right\_link = p;

p->right\_link = a\_right;

p->left\_link = a;

a\_right->left\_link = p;

p = parent;

}

free(del\_node);

}

}

//关键值减值操作

void decrease(fiboheap \*&a, int key, int value)

{

fiboheap \*p = find(a, key);

if(p == a) {//无对应的节点

p->data.key -= value;

} else if(p->child == false) {//所操作节点处于顶点节点组成的双向循环链表中

p->data.key -= value;

if(p->data.key < a->data.key) {

a = p;

}

} else {

fiboheap \*q = p;

p->data.key -= value;

//级联剪枝操作

while(p->child\_cut != false) {

fiboheap \*parent = p->parent;

p->parent = NULL;

fiboheap \*p\_left = p->left\_link;

fiboheap \*p\_right = p->right\_link;

p\_left->right\_link = p\_right;

p\_right->left\_link = p\_left;

p->child\_cut = false;

parent->child = p\_right;

parent->degree--;

fiboheap \*a\_right = a->right\_link;

a->right\_link = p;

p->right\_link = a\_right;

p->left\_link = a;

a\_right->left\_link = p;

p = parent;

}

if(q->data.key < a->data.key) {

a = q;

}

}

}

element del\_min(fiboheap \*&a)

{

element temp = a->data;

fiboheap \*tree[TREELEN];

for(int i=0; i<TREELEN; ++i) {

tree[i] = NULL;

}

fiboheap \*p1, \*p2;

if(a->left\_link != a) {

p1 = a->left\_link;

p1->right\_link = a->right\_link;

a->right\_link->left\_link = p1;

}

p2 = a->child;

fiboheap \*m1 = p1;

fiboheap \*m2 = p2;

do {

if(m1 == NULL) break;

// printf("%d\n", m1->data.key);

int degree;

fiboheap \*w = m1->right\_link;

for(degree=m1->degree; tree[degree]; ++degree) {

join\_min\_trees(m1, tree[degree]);

tree[degree] = NULL;

}

tree[degree] = m1;

m1 = w;

} while (m1 != p1);

do {

if(m2 == NULL) break;

// printf("%d\n", m2->data.key);

int degree;

fiboheap \*w = m2->right\_link;

for(degree=m2->degree; tree[degree]; ++degree) {

join\_min\_trees(m2, tree[degree]);

tree[degree] = NULL;

}

tree[degree] = m2;

m2 = w;

} while (m2 != p2);

int k = 0;

for(int i=0; i<TREELEN; ++i) {

if(tree[i] == NULL) continue;

k++;

if(k == 1) {

tree[i]->right\_link = tree[i]->left\_link = tree[i];

a = tree[i];

} else if(k > 1) {

fiboheap \*temp = a->right\_link;

a->right\_link = tree[i];

temp->left\_link = tree[i];

tree[i]->right\_link = temp;

tree[i]->left\_link = a;

if(a->data.key > tree[i]->data.key) a = tree[i];

}

}

return temp;

}

void insert(fiboheap \*&a, fiboheap \*b)

{

if(a == NULL) {

a = b;

return;

}

fiboheap \*temp = a->right\_link;

a->right\_link = b;

b->right\_link = temp;

temp->left\_link = b;

b->left\_link = a;

if(a->data.key > b->data.key) {

a = b;

}

}

void merge(fiboheap \*&a, fiboheap \*b)

{

if(a == NULL) {

a = b;

return;

}

fiboheap \*p1 = a->right\_link;

fiboheap \*p2 = b->left\_link;

a->right\_link = b;

b->left\_link = a;

p1->left\_link = p2;

p2->right\_link = p1;

if(a->data.key > b->data.key) {

a = b;

}

}

void traverse(fiboheap \*a)

{

fiboheap \*p = a;

do {

printf("%4d%4d%4d", p->data.key, p->degree, p->child\_cut);

if(p->child != NULL) printf(" 1 ");

else printf(" 0 ");

if(p->parent != NULL) printf(" 1 ");

else printf(" 0 ");

printf("\n");

if (p->child != NULL) {

traverse(p->child);

}

p = p->right\_link;

} while(p != a);

}

int main()

{

fiboheap \*a = NULL;

for(int i=1; i<=8; ++i) {

fiboheap \*p = new fiboheap;

p->right\_link = p->left\_link = p;

p->child = p->parent = NULL;

p->data.key = i;

p->degree = 0;

p->child\_cut = false;

insert(a, p);

}

fiboheap \*b = NULL;

for(int i=9; i<=17; ++i) {

fiboheap \*p = new fiboheap;

p->right\_link = p->left\_link = p;

p->child = p->parent = NULL;

p->data.key = i;

p->degree = 0;

p->child\_cut = false;

insert(b, p);

}

del\_min(a);

del\_min(b);

merge(a, b);

traverse(a);

printf("\n");

decrease(a, 13, 15);

// del\_fix(a, 13);

traverse(a);

}

/\*

class Point //Point 类的声明

{

public: //外部接口

Point(int xx=0, int yy=0) {X=xx;Y=yy;} //构造函数

Point(Point &p); //拷贝构造函数

int GetX() {return X;}

int GetY() {return Y;}

private: //私有数据

int X,Y;

};

//成员函数的实现

Point::Point(Point &p)

{

X=p.X;

Y=p.Y;

cout<<"拷贝构造函数被调用"<<endl;

}

//形参为Point类对象的函数

void fun1(Point p)

{ cout<<p.GetX()<<endl;

}

//返回值为Point类对象的函数

Point fun2()

{

Point A(1,2);

return A;

}

//主程序

int main()

{

Point A(4,5); //第一个对象A

Point B(A); //情况一，用A初始化B。第一次调用拷贝构造函数

cout<<B.GetX()<<endl;

fun1(B); //情况二，对象B作为fun1的实参。第二次调用拷贝构造函数

// B = fun2(); //情况三，函数的返回值是类对象，函数返回时，调用拷贝构造函数

cout<<B.GetX()<<endl;

return 0;

}

\*/