左偏树(指针实现)

简介

左偏树是一种**可并堆**，核心操作是 $O(\log n + \log m)$ 的 merge 操作。

通过 merge 操作来实现 push 和 pop 操作。

基本信息

- 左偏树的结构是**二叉树**。
- 每个节点具有一个额外属性 dist。定义一个节点是**边缘节点**，当且仅当它的**儿子个数不为 2**。dist 表示**该节点往儿子方向走，走到边缘节点需要经过的最小边数**，空节点 dist 定义为 -1。
- 左偏树每个节点的左子树的 dist **不小于**右子树的 dist，所以显然有 `dist = right_son->dist + 1`。

算法流程

push

创建一个新的堆，只分配一个节点，将新堆合并进原有堆。

pop

原有根节点删去，合并其左右儿子，得到新的根节点。

merge

1. 找到根节点 val 较小的那个堆，将它的根节点作为新堆的根节点，它的左儿子作为新堆的左儿子，它的右儿子和另一个堆合并，作为新堆的右儿子。
2. 合并时遇到一个堆为空时，非空堆即合并结果，可以直接返回。
3. 如果合并后右儿子的 dist 大于左儿子的 dist，交换两个儿子。

复杂度分析

结论：

- push：$O(\log n)$
- pop：$O(\log n)$
- merge：$O(\log n + \log m)$

证明：

1. 若一个节点的 dist 为 x，则它及其子树至少有 2^x 个节点。故 dist 的值是 $O(\log n)$ 级别的。
2. 进行合并时，每递归一层，参与合并的**两个堆的 dist 之和减少 1**，故递归层数为 $O(\log n + \log m)$。

实现

结构

- 一个内部的类 `Node` 表示左偏树的一个节点，包含 `dist`， `val` 两个变量，以及 `son[0], son[1]` 两个指针，分别指向左儿子和右儿子。
- 一个类 `Heap`，表示一个堆，包含一个 `Node` 指针，表示根节点。

声明

这里希望练习使用 C++11 中的**智能指针**和**移动语义**，所以声明为：

```
class Heap{
private:
    class Node{
    public:
        unique_ptr<Node> son[2];
        int val, dist;
        Node(int val=0){
            this->son[0]=nullptr;
            this->son[1]=nullptr;
            this->val=val;
            this->dist=0;
        }
    };
    unique_ptr<Node> root;
public:
/*
    其它部分暂时省略
*/
};
```

构造函数

```
class Heap{
    Heap(unique_ptr<Node> root){
        this->root=move(root);
    }
};
```

通过传入一个 `unique_ptr<Node>`，构造一个 `Heap` 对象。

`unique_ptr` 是一种智能指针，它指向的对象是它独享的，不能被其它东西访问。

`unique_ptr` **仅支持移动语义**，这用于**转移**对象的所有权，对象所有权转移后，原来的 `unique_ptr` 将失效。

`unique_ptr` **不支持拷贝和赋值**操作，所以上面的构造函数严格来说是有问题的，如果传入的 `unique_ptr` 不是右值则会**尝试调用不存在的拷贝构造函数**，导致编译错误。故应该将参数声明为**右值引用**(注意区分**右值引用**和**常值引用**)。

具体如下：

```
class Heap{
    Heap(unique_ptr<Node> &&root){
        this->root=move(root);
    }
};
```

至于为什么第一份代码是正确的，是因为只要保证传入的 `unique_ptr` 是右值，编译器就会**优先自动调用移动构造函数**，所以不需要在调用函数时显式地写出 `move`。

至于为什么第二份代码中**右值引用**的 `root` 仍然需要使用 `move` 来显式的转化为右值，这是因为**右值在绑定到一个右值引用后，其本身在作用域内是一个具名变量，行为会退化退化为左值。**，故需要显式的使用移动语义来调用移动构造函数。

merge

```
class Heap{
    Heap& merge(Heap &&other){
        if(root==nullptr){
            root=move(other.root);
        }
        else if(other.root==nullptr){

        }
        else{
            if(root->val > other.root->val){
                swap(this->root, other.root);
            }
            root->son[1] = move(Heap(move(root-
>son[1])).merge(move(other.root)).root);
            if(root->son[0] == nullptr ||
                (root->son[1] != nullptr && root->son[0]->dist < root->son[1]-
>dist)){
                swap(root->son[0], root->son[1]);
            }
            if(root->son[1] != nullptr)
                root->dist = root->son[1]->dist + 1;
            else
                root->dist = 0;
        }
        return *this;
    }
};
```

merge 返回一个**左值引用**是为了方便链式调用。

有几个细节：

- `swap`：可以用于交换 `unique_ptr`，它应该是实现了这种移动语义的交换。

- 隐式构造: `Heap(xxx).merge(move(other.root)).root` 这里使用了**隐式构造**，`Heap.merge` 支持的参数是一个右值的 `Heap`，传入时传的一个右值的 `unique_ptr`，由于定义了右值 `unique_ptr` 到 `Heap` 的构造函数，这里直接**隐式调用**了构造函数，构造了一个**右值的** `Heap` 并作为参数传递给了 `merge`。
- 返回左值引用: `return *this` 返回左值引用的目的是方便**链式调用**，至于为什么临时构造的 `Heap` 能返回一个左值引用，是因为**临时对象**的**生存周期**是到**表达式结束**为止，而**临时对象在生命周期内可以被左值引用。**

push && pop && top

```
class Heap{
    int top() const{
        return root->val;
    }
    void push(int x){
        this->merge(unique_ptr<Node>(new Node(x)));
    }
    int pop(){
        this->root = move(Heap(move(root->son[0])).merge(move(root->son[1])).root);
        // 当 merge 函数返回时，它返回的是临时对象的左值引用，临时对象在生命周期内可以被左值引用.
        return top();
    }
};
```

比较简单, 不解释了.

左值右值的理解

简单的说：

- 右值是**只能放到表达式右边的值**，左值是**既能放到表达式右边，也能放到左边的值。**
- 右值一般是**没有命名的值**，左值一般是**有名字的值。**
- 右值是**临时的值**，左值是**持久的值。**

更具体一点：

左值的特性

- 有固定内存地址：左值通常存储在堆或栈上，可以通过取地址操作符（&）获取其地址。
- 可以被多次访问：左值的生命周期较长，可以在多个语句中被访问。
- 可以被修改：左值通常可以被修改，例如变量赋值操作。

右值的特性

- 没有固定内存地址：右值通常是临时对象，没有**固定的存储位置**，或者其存储位置在表达式结束后立即失效。
- 不能被取地址：右值不能使用取地址操作符（&）获取其地址。
- 不能被多次访问：右值的生命周期仅限于当前表达式，不能被多次访问。

- 通常不可修改：右值通常是不可修改的，因为它们是临时的。

右值举例

C++ 的值除了右值就是左值，下面几个右值的例子：

- 函数的返回值是一个右值。
- 字面量，`10`，`"hello"` 是右值。
- 临时对象是右值，例如 `c = a + b;` 中的 `a + b` 这个整体。

下面是左值的例子：

- 命名变量名是左值，例如 `int a = 10;` 中的 `a`，`a = b = c = 10;` 的 `a, b, c` 都是左值。

左值引用和右值引用

名词解释

```
int a = 10; // 左值 被赋值为 右值 , a 是左值, 10 是右值.
int &&x1 = 10; // 右值引用 绑定到 右值, x1 是右值引用, 10 是右值, 允许.
int &&x2 = a; // 右值引用 绑定到 左值, x2 是右值引用, a 是左值, 允许, 一般用于模板编程.
int &y = a; // 左值引用 绑定到 左值, y 是左值引用, a 是左值, 允许.
int &y1 = 10; // 左值引用 绑定到 右值, y1 是左值引用, 10 是左值, 编译错误.
int &y2 = x1; // 左值引用 绑定到 右值引用, y2 是左值引用, x1 是右值引用, 编译错误.
int &&z1 = y; // 右值引用 绑定到 左值引用, z1 是右值引用, y 是左值引用, 允许, 并且, 支持引用折
叠语法, 所以 z1 的实际行为是左值引用.
int &z2 = z1; // 左值引用 绑定到 可折叠为左值的 右值引用, z2 是左值引用, z1 是右值引用, 但可折
叠到对 a 的左值引用, 允许.
int &z3 = x2; // 左值引用 绑定到 可折叠为左值的 右值引用, z3 是左值引用, x2 是右值引用 但可折叠
到对 a 的左值引用, 允许.
```

右值引用的特性

右值引用似乎是**命名变量**吧?

所以：右值引用可以在不发生内存操作的前提下将**临时右值变为左值**，可以延长右值的生命周期，降低读写内存的开销。

左值引用的特性

左值引用等价于为变量取了一个别名。

特别的，在函数传参如果使用左值引用传参，函数类通过左值引用可以修改原变量的值。

引用折叠

在不发生编译错误的情况下, 可以支持引用折叠, 名词解释中的 `z1, z2, z3` 例子解释了这个特性。

具体的，在发生多次引用绑定时，按照以下规则折叠：

- 左 + 左 = 左
- 右 + 左 = 左

- 左 + 右 = 左
- 右 + 右 = 右

完整代码

```
#include "bits/stdc++.h"

using namespace std;

class Heap{
private:
    class Node{
    public:
        unique_ptr<Node> son[2];
        int val, dist;
        Node(int val=0){
            this->son[0]=nullptr;
            this->son[1]=nullptr;
            this->val=val;
            this->dist=0;
        }
    };
    unique_ptr<Node> root;
public:
    Heap(unique_ptr<Node> &&root){
        this->root=move(root);
    }
    int top() const{
        return root->val;
    }
    void push(int x){
        this->merge(unique_ptr<Node>(new Node(x)));
    }
    Heap& merge(Heap &&other){
        if(root==nullptr){
            root=move(other.root);
        }
        else if(other.root==nullptr){

        }
        else{
            if(root->val > other.root->val){
                swap(this->root, other.root);
            }
            root->son[1] = move(Heap(move(root-
>son[1])).merge(move(other.root)).root);
            if(root->son[0] == nullptr ||
                (root->son[1] != nullptr && root->son[0]->dist < root->son[1]-
>dist)){
                swap(root->son[0], root->son[1]);
            }
```

```
            if(root->son[1] != nullptr)
                root->dist = root->son[1]->dist + 1;
            else
                root->dist = 0;
        }
        return *this;
    }
    int pop(){
        this->root = move(Heap(move(root->son[0])).merge(move(root->son[1])).root);
        // 当 merge 函数返回时，它返回的是临时对象的左值引用，临时对象在生命周期内可以被左值引用.
        return top();
    }
};

int main(){
    Heap h(nullptr);
    int n;
    cin >> n;
    for(int i=1;i<=n;i++){
        int tp, x;
        cin >> tp;
        switch (tp){
            case 1:
                cin >> x;
                h.push(x);
                break;
            case 2:
                cout << h.top() << endl;
                break;
            case 3:
                h.pop();
                break;
            default:
                throw "unsupported operation";
        }
    }
    return 0;
}
```