# Peter 笔记

Peterlits Zo

2020 年 7 月 13 日

## 目录

**1 过长的 LaTeX url** **2**

**2 并查集** **2**

2.1 引入 . . . 2
2.2 我的想法 . . . 2
2.3 优化 . . . 2
2.4 关于并查集的实现 . . . 2
2.5 改进的方法与一些想法 . . . 3

**3 C++ 中关于 set<set<int>> 的一些小问题** **3**

3.1 关于 const 的 set::iterator . . . 3
3.2 关于 set 的方法 extract . . . 4
3.2.1 函数 std::move . . . 4
3.2.2 Node Handle . . . 4
3.2.3 关于 extract . . . 4

**4 C++ 中的 set 字面量** **5**

**5 在 C++ 循环时删除元素** **5**

**6 LeetCode-1025 题解** **5**

6.1 题干 . . . 6
6.2 题解 . . . 6

# 1 过长的 LaTeX url

有的时候 LaTeX 的 url 会很长很长，但是并不会换行。

在 Stack Overflow[1]中指出，`url` 宏包定义了`\UrlOrds` 如下：

```
\def\UrlOrds{\do\*\do\-\do\~\do\'\do\"\do\-}%
```

我们可以通过 `\g@addto@marco`，来将他们添入可以换行的宏中：

```
\g@addto@macro{\UrlBreaks}{\UrlOrds}
```

以上即可定义换行的地方。

# 2 并查集

## 2.1 引入

并查集的概念是在做一个题目的时候引入近来的，题目大体是：有一个集合 $S$，其中告知从属于一系列从属于子集合的两个元素的序列，求最多会有多少个子集合?

我们不妨这么想：如果告知 $S = \{1, 2, 3, 4\}$ 其中 1 和 2 为一个子集合的两个，2 和 3 是一个子集合的元素，那么很明显 $1, 2, 3$ 属于同一个子集合，最多有两个子集合。

## 2.2 我的想法

一开始我想的是用 `set<set<int>>` 来表达这个数据结构，一开始让所有的元素都从属于自己的一个子集合，那么明显有，每次告知 $i, j$ 属于一个集合的时候，就遍历一遍大的 `set<set<int>>` 如果有一个集合含有 $i$ 或者 $j$ 的话，就把它拖入到一个 `temp` 的 `set<int>` 数据结构中，然后把原来的两个集合替换成 `temp`。

很明显，对于一个 $i, j$ 数据的输入的话，那么它需要复杂度为 $O(S.len)$ 的时间复杂度。

## 2.3 优化

在最初的版本中我们可以看到每一次确定集合到底在哪里都需要花很长很长的时间。所以说我们使用一个 `map<int, set<int>*>` 的数据结构来表示它的对应的结构，这也算是百分百的 map 的。但是即便如此，我们在进行集合的归并的时候仍然需要花费不少的时间，看来我们更需要一个自递归的数据结构来表示并查集。这样，我们能轻松的将两个数据合并为一个数据结构，而且同时不需要进行内存的重新分配。这样，树就自然而然的出来了。

## 2.4 关于并查集的实现

我们知道，并查集中最重要的是：知道一个数据属于哪一个集合，以及合并两个集合。所以说我们有下面的伪代码，首先是定义基本的树结构来表示并查集：

[1]更多信息参见：`https://stackoverflow.com/review/suggested-edits/26648630`。

```
struct Tree {
    Tree* parent;
}
map<int, Tree> trees;
```

其中我们使用 map<int, Tree> 来快速地寻找对应的集合（形式是树[2]）。而我们如何定义两个元素属于一个集合呢？那就是他们的 parent 相同，同时由于树结构的性质，我们可以在一个小复杂度时间内把一个集合融入另一个集合中，也就是说把它的 parent 的指针指向另一颗树的随便的一个子元素就行了，有代码：

```
Tree* parent(Tree* t) {
    if (t->parent == t) {
        return t;
    } else {
        return parent(t->parent);
    }
}

void union(Tree* a, Tree* b) {
    parent(a)->parent = parent(b);
}
```

最后查询有多少个集合的时候，只需要知道有多少个自己是自己的 papent 的元素就行了。

### 2.5 改进的方法与一些想法

因为和其他的树不同，这个树只能从元素知道它的根，所以降低树的高度就是直接把元素挂靠在根元素上就行了。所以说一般而言不会因为树太高而导致复杂度上升，但是也可以考虑一点，那就是依次线性的挂靠上去，而最终看起来更像一个线性链表。红黑树的实现方法最终还是靠了解直接点搞上去的，要不然让每一个元素都挂靠到根节点上去，这样的复杂度在前期又太高了。

## 3 C++ 中关于 set<set<int>> 的一些小问题[3]

在尝试 AC 一个关于查并集的题目的时候，我使用了一个 set<set<int>> 的数据结构，但是在之后我发现了我无法在迭代中修改 set<int> 类型的 element。

### 3.1 关于 const 的 set::iterator

因为 set 是有序的，好像底层是用红黑树搞的。所以说在迭代的时候是不能修改的，不然底层就乱套了。可是这对于一些代价小的数据，比如 int 而言直接 erase 和一个 insert 就没事了，但是数据本身是 set<int> 类型的，重新搞一个代价就会很大。

[2]其实在 C++ 中也是用红黑树来表示集合的，所以说用树来表示也没有什么不好的!

[3]更多详情请参见：https://stackoverflow.com/questions/62847368/why-i-cannot-change-the-value-in-loop-of-setsetint。这个网址是我在 Stack Overflow 上提的问题。

## 3.2 关于 set 的方法 extract[4]

在了解 extract 之前，了解 std::move 也是很有必要的。

### 3.2.1 函数 std::move

我之前使用过一个关于 json 的库，在更之前的时候也看过一点点关于 Rust 的书，其中移动是一个很重要的概念，它改变了内存的所有者。但是因为不需要复制了，所以带来了很显著的性能提升，这太棒了啦。

根据一篇很不错的博客[5]所说，我们可以有一个很糟糕的例子来说明为什么我们需要 std::move：

```
template<typename T> swap(T& a, T& b) {
    T tmp(a);       // now we have two copies of a
    a = b;          // now we have two copies of b
    b = tmp;        // now we have two copies of tmp (aka a)
}
```

接下来是一个好的例子：

```
template<typename T> swap(T& a, T& b) {
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

其中 move 函数的底层逻辑是这样的：

```
template <typename T>
typename remove_reference<T>::type&&
move(T&& a) {
    return a;
}
```

看起来，移动语义的底层是右值引用。这很不错的！关于右值引用和移动语义的更多话题，请多多看看之前提到的博客。

### 3.2.2 Node Handle[6]

extract 会返回一个 node handle，所以先了解一哈什么是 node handle 也是很有必要的哦。node handle 是一个尽可移动的值，它提供了处理 node 的一系列接口，比如 value() 函数会提供子对象（也就是说在里面的那个对象）的引用。对于 map 而言，key() 也会很让它开心。主要而言这就是基本的引用了，一个高抽象的 handle。

### 3.2.3 关于 extract

extract 函数主要是首先断开 node 和容器的连接，然后返回一个 handle，所以说，接下来就可以使用 handle 来进行操作。

比如说：

[4]更多信息参见：https://en.cppreference.com/w/cpp/container/set/extract。

[5]更多详情请参见：http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html#Move_Semantics。

[6]更多详情请参见：https://en.cppreference.com/w/cpp/container/node_handle。

```
for(set<set<int>>::iterator it = test.begin(); it != test.end(); ++it) {
    auto nh = test.extract(it);
    nh.value().insert(100);
    test.insert(move(nh));
}
```

# 4 C++ 中的 set 字面量

set 的字面量本来是可以写成 `set<int>{1, 2, 3}`，也就是说类型后面接上一个初始化列表，但是好像有一些是不支持初始化列表的。比如说 C++11 之前的。但是在 C++ 之前也可以使用初始方法来调用，比如说 `vector<int>` 就支持 `vector<int>(10)`来当作它的字面量。

在 C++11 之前的话，有三种比较 nice 的初始化方法：

1. 空。初始化一个空列表。
2. 范围初始。使用一个 array 或者可以可以迭代的来进行迭代复制。
3. 拷贝初始。使用一个 set 来直接复制一份。

所以说有的时候我们可以用这种来初始化一个 `set<int>`。

```
int temp[] = {1, 2, 3, 4, 5};
set<int> target(temp, temp + sizeof temp / sizeof *temp);
```

# 5 在 C++ 循环时删除元素

在 `vector` 中，方法 `erase` 会删除现有的元素，然后返回它的下一个元素的迭代器，而这个是最好的方法，即：

```
it = test.erase(it);
```

有的时候，在循环的时候进行操作是一个很危险的事情，除非你真的清楚你在干什么，不然最好还是慎重进行。无论是无效化迭代器还是诸如此类的事情，都开始变得越来越危险了。

# 6 LeetCode-1025 题解

这是一道非常非常简单的题，但是很有意思，特此记下。

## 6.1 题干

爱丽丝和鲍勃一起玩游戏，他们轮流行动。爱丽丝先手开局。

最初，黑板上有一个数字 $N$。在每个玩家的回合，玩家需要执行以下操作：

选出任一 $x$，满足 $0 < x < N$ 且 $N \mod x == 0$。用 $N - x$ 替换黑板上的数字 $N$。如果玩家无法执行这些操作，就会输掉游戏。

只有在爱丽丝在游戏中取得胜利时才返回 `true`，否则返回 `false`。假设两个玩家都以最佳状态参与游戏。

## 6.2 题解

这道题开始一分析，就是两个小伙伴用能整除 $N$ 的数 $x$ 依次减去得到新的 $N$，但是很容易发现 $N$ 因为其复杂的分解属性，所以对于他对应的 $x$ 也显得很复杂。

大家都知道博弈题要用归纳法来做对吧，那归纳如表1。

| Number | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
|---|---|---|---|---|---|---|---|---|---|
| If win | F | T | F | T | F | T | F | T | ... |

表 1: 关于先手得到数之后到底时候能赢的归纳表

我们可以看到，先手能赢的必要条件就是后手必输。对于归纳表的下一个数，如果它是奇数的话，那么所有的 $x$ 都是奇数，所以后手得到的数则一定是偶数，后手必赢，那它先手必输。如果它是偶数的话，$x$ 可以取 1、也可以取偶数、或者可能的奇数，下一个数可以是奇数，后手必输，那么它先手必赢。

可以看到无论在基本情况还是递归归纳情况下，始终保持着偶数胜利奇数失败的传统，所以就可以定言 — 先手偶数则胜利，先手奇数则失败。