

一些因缺思汀的小实验

不同代码实现方式效率上的差异



cjsoft

F**k me on github!

<https://github.com/cjsoft>

Summary

打星号的没做辣QAQ



讲道理

TEST1 使用new和delete的常数开销与Buffer开销的对比

vector

TEST2 使用指针跳转和数组跳转的常数对比

map

Buffer+指针

TEST4 *手写垃圾回收的实现思路（这专题没啥卵用）

new+指针

TEST5 *STL string和char string的小细节（也没啥卵用）

TEST3 常用基础数据结构STL和手写版常数对比

TEST6 fill_n函数、memset函数、传统for赋值的对比

stack

TEST7 linux下的对拍

priority_queue(vector)

实验环境



TEST1 使用new和delete的常数开销与Buffer开销的对比

new和delete是c++中在堆上申请和释放内存的操作符（废话）

为了让大家对这两个东西的常数有一些具体的认识，我做了个小测试。

测时方法：clock

测试代码及数据生成脚本：<https://code.csdn.net/cyh19990309/noip/tree/master/OI-related/speech/tests/test1>

读入数据时间不计

TEST1 使用new和delete的常数开销与Buffer开销的对比

操作	次数	开销/CLOCK
new 一个 int 并初始化	10,000,000	296400
delete 一个 int	10,000,000	116795
从buffer中申请并初始化一个int	10,000,000	33689

new的时间开销有buffer的将近9倍！、

这非常容易理解，因为buffer的内存是连续的，早早申请好的，时间开销近乎可以忽略。

而new的操作过于碎片化，每次只申请4个字节，却要申请1kw次，时间开销不可忽略。

---分割线---

什么时候用new？

一般在写一些奇奇怪怪用指针乱搞的数据结构的时候会用到new来动态申请内存，但几乎所有情况下都可以改用buffer进行。

那么既然用new时间开销这么大，是不是全用buffer就可以了？

也不是绝对的，因为有new就有delete。new完的内存可以释放，而buffer不可以，一般开buffer都会超过1MB，你不可能在函数内部开buffer，这样就导致了buffer没有用了之后所占用的内存不能释放，在内存较为紧张的时候，这是比较危险的。

TEST1 使用new和delete的常数开销与Buffer开销的对比

操作	次数	开销/CLOCK
new 一个 int 并初始化	10,000,000	296400
delete 一个 int	10,000,000	116795
从buffer中申请并初始化一个int	10,000,000	33689

那么怎么搞才是最好呢？、

那当然是改new一个int变为new一个buffer，

这样二者的优点都可以有，缺点都可以没有

buffer用完之后之后delete[]就好了

TEST2 使用指针跳转和数组跳转的常数对比

指针跳转和数组跳转是两种常用的小玩意，一般链前啦，二叉搜索树啦，链表啦，都可能会涉及到选择这两个之中哪一个的问题

为了让大家对这两个东西的常数有一些具体的认识，我做了个小测试。

测时方法：clock

测试代码及数据生成脚本：<https://code.csdn.net/cyh19990309/noip/tree/master/OI-related/speech/tests/test2>

读入数据时间不计

TEST2 使用指针跳转和数组跳转的常数对比

操作	次数	开销/CLOCK
buffer+指针跳转	10,000,000	91606
数组跳转	10,000,000	92324
用new申请的元素+指针跳转 (new时间不计)	10,000,000	405035

我的数据是绝对随机的，但是只保证没有自环

使用了buffer之后，指针还是要比纯数组要快一丢丢的

因为new申请的元素位置比较散乱，所以4倍多的常数容易理解

由此可见，在相近内存区域间，单纯使用operator[n]进行找位置再跳转的效率与使用指针进行跳转的效率差异不大，我不觉得会因为使用或者不使用指针进行跳转，一道题目的结果会因此有所改变。

他们之间带来的更多差异是代码风格上的，有人觉得用数组比较舒服，有人觉得用指针比较舒服，这都无妨

(或许是因为随机出来的数据太水了？)

TEST3 常用数据结构的手写实现和STL实现的对比

目前我做了stack、priority_queue和vector

map因为时间不够了所以没做，queue和stack类似

为了让大家对这三个东西的常数有一些具体的认识，我做了个小测试。

测时方法：clock

测试代码及数据生成脚本：<https://code.csdn.net/cyh19990309/noip/tree/master/OI-related/speech/tests/test3>

读入数据时间不计

TEST3 常用数据结构的手写实现和STL实现的对比

操作	次数	开销/CLOCK	VmRSS/KB	VmPeak/KB
12行手写stack<int>	10,000,000	517068	15724	43400
std::stack<int>	10,000,000	633878	16056	26312

我的数据是绝对随机的，但保证push:pop=2:1

从时间上来看，std::stack稍微大一些，大约1.2倍左右。

从内存开销上来看，常驻内存差别不大，但内存峰值有较大差别。

大概是因为std::stack的内存分配是动态的？

这也容易理解为什么std::stack的时间开销会比手写stack大一些。

TEST3 常用数据结构的手写实现和STL实现的对比

操作	次数	开销/CLOCK
74行手写heap<int> extract	1,000,000	1144778
std::priority_queue<int, vector<int> > pop	1,000,000	1070153

数据是绝对随机的，但保证都是正整数

从时间上来看，std::priority_queue的pop反而更快一些（大约7w多个clocks）

但考虑priq的diy空间不大，而且不能O(n)的初始化或者修改q中的元素

窃以为手写heap不知道比priq要高到哪里去了

当然如果神马强制在线题不能heapify啥的用priq还是很不错的选择哒

TEST3 常用数据结构的手写实现和STL实现的对比

操作	次数	开销/Sec
vector<int>.push_back(int)	100,000,007	1.439846
vector iterating using iterator	100,000,007	2.136002
vector iterating without reconstructing tail iterator	100,000,007	1.389050
vector Iterating by operator[]	100,000,007	0.688580
array iterating	100,000,007	0.294152

第二行的那个写法是ST里默认的snippet forv，感受一下、、、

恩，如果用vector的话，枚举果断用operator[]，用iterator果然是找死（足足是operator[]的两倍啊喂）

恩，vector.operator[]又是数组枚举的2倍多、

但鉴于vector是懒癌患者的福音，什么时候用，什么情况下用，就是仁者见仁智者见智咯、

恩，反正我以前因为用vector被卡掉过一次

TEST3 常用数据结构的手写实现和STL实现的对比

操作	次数	开销/CLOCK	VmRSS/KB	VmPeak/Kb
130行手写splay map<int, int> insert	1,000,000	3027259	40412	43404
130行手写splay map<int, int> query	333,333	1167679		
std::map<int, int> insert	1,000,000	2133524	49228	59536
std::map<int, int> find	333,333	616932		

本例中开销包括读入时间，内存测量保证较精确

RBTree碾压啊、

手写的spt因为查询和插入操作类似，所以开销与操作次数近乎成正比

大概因为我spt写的太丑辣、常数这么大

不过还是鉴于手写的平衡树DIY空间大、能够精确控制内存占用（雾）内存占用稍小一点？

这个实验告诉我们：如果你会写些更搞基的平衡树、就不用纠结这些东西了、

TEST6 fill_n函数、memset函数、传统for赋值的对比

三种都是用来初始化数据的方法

测时方法：clock

测试代码及数据生成脚本：<https://code.csdn.net/cyh19990309/noip/tree/master/OI-related/speech/tests/test6>

读入数据时间不计

TEST6 fill_n函数、memset函数、传统for赋值的对比

操作	次数	开销/CLOCK
fill_n 一个 int 数组为 -1	100,000,000	238054
memset 一个 int 数组为 0	100,000,000	34822
for 赋值一个 int 数组为 -1	100,000,000	254354

可以看出memset的速度最快、一般大家也都是用它。

但是memset的局限性也比较多，不能赋任意初值，如果是自定义struct的批量初始化就gg了。

fill_n比for要快一点点。

但是毕竟fill_n只要一行一句话，for要三行啊

恩，还有一个函数，叫fill，他比for还要慢一点，并不知道为什么

TEST7 Linux下的对拍

随手写的并不因缺思汀的话题

首先你需要先造输入数据、

正常写

然后你需要暴力出结果

正常写

标准流重定向大家都会用吧 < >

数据造好之后

```
import os
```

```
for i in xrange(1, 10 + 1):
```

```
    os.system('diff -Bb asd%d.in asd%d.out > result%d.txt')
```

恩，反正我觉得这玩意看起来比bash script舒服。

之后查下result就知道对拍结果了

其实和win下的对拍基本一样、、
只不过不是batch script
比较文件不是用fc而已、

讲道理



讲道理



讲道理



讲道理



讲道理

讲道理



讲道理



讲道理



讲道理

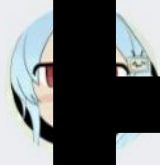


讲道理

讲道理



讲道理



讲道理



讲道理



讲道理

讲道理



讲道理



讲道理



讲道理



讲道理

讲道理



讲道理



讲道理



讲道理



讲道理

TH

X!