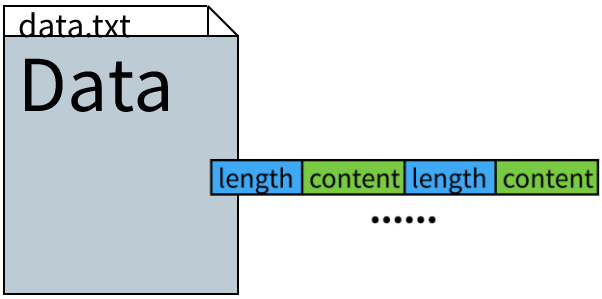
简易数据存储系统

—— 朱超捷 5130379013

总体架构

整个数据存储系统由三个文件组成，分别是 index.txt, data.txt, idle.txt 他们各自的作用如下

数据文件 data.txt

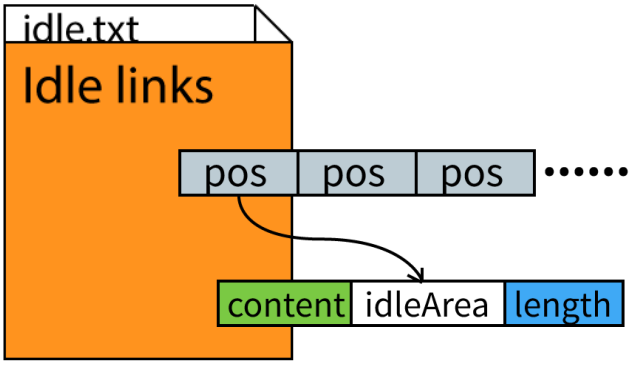


以二进制的形式，连续存储数据文件的数据，以如下结构存储：

长度 + 内容

这里之所以要存储长度，是为了之后指向数据文件空闲区域的 idle 指针能够有更高的空间效率，之后会进行介绍。

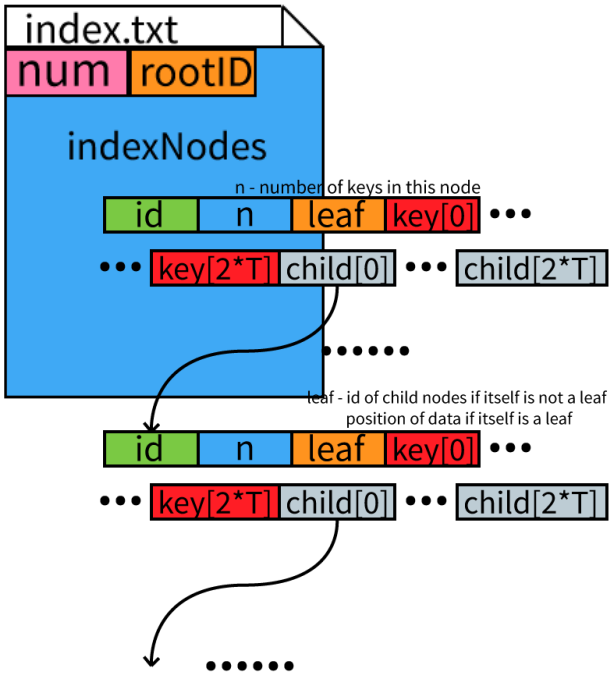
空闲区块文件 idle.txt



以二进制的形式，连续存储 pos 值。每个 pos 值代表从数据文件 data.txt 的第 pos 个 byte 开始的长度为*(pos)的数值是过期的、失效的。

当程序读入 idle.txt 时，会生成一个根据长度排序的链表，每个新的结点通过插入排序的方法插入链表。当用户需要插入新的数据时，会从头扫描链表找到一个大于长度的最小长度结点，从而达到 best fit 效果，进而提高 data.txt 的空间使用效率。

索引文件 index.txt

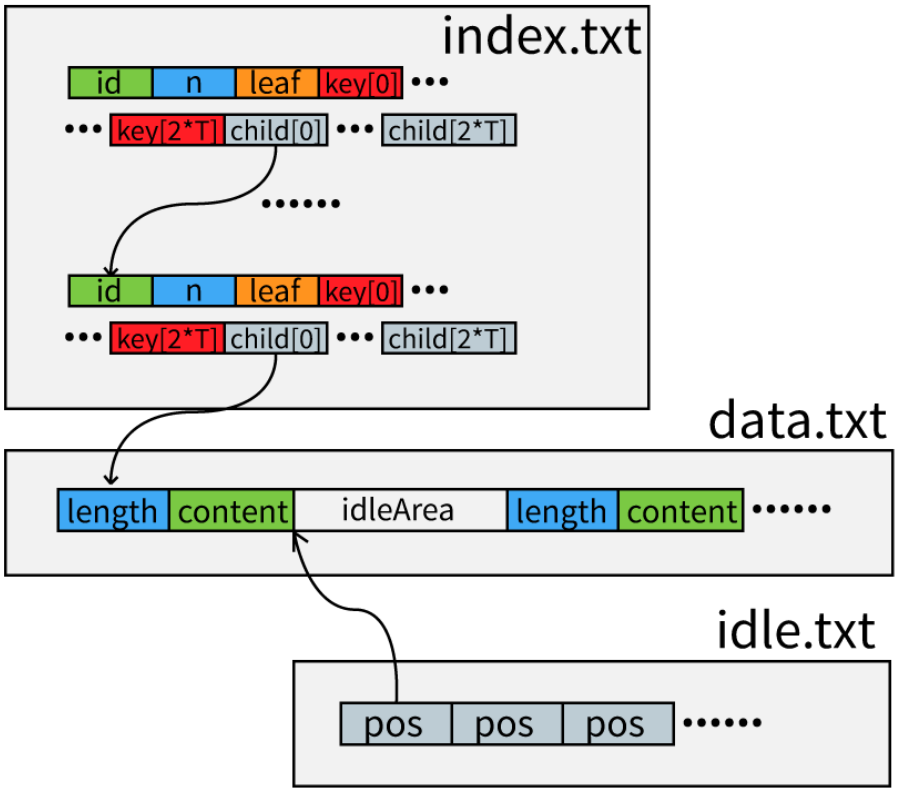


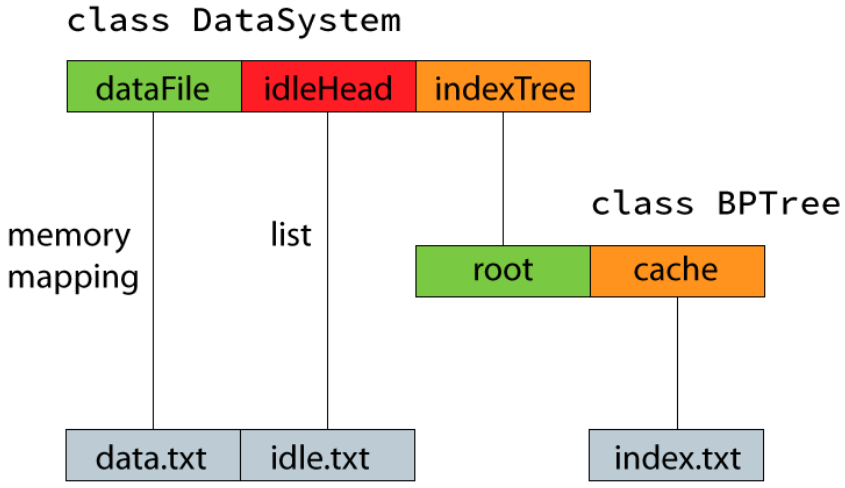
以二进制的形式，连续存储作为索引的 B+树结点。在 B+树结点之前，先要存储 num 和 rootID，前者为下次插入确定新位置做准备，后者为下次查找和删除操作做准备。每个 B+树结点以如下结构存储：

$$\mathrm{id} + \mathrm{n} + \mathrm{leaf} + \mathrm{key}[0..2 * \mathrm{T} - 1] + \mathrm{child}[0..2 * \mathrm{T} - 1]$$

其中各项的意义，将在 B+树具体数据结构中进行介绍。

最后，我们来总览一下整个数据存储系统的总体文件之间的联系和结构：
而其实际在程序中控制读入内存的情况如下：





我一共设计了两个类来管理整个数据存储系统，其中一个是 DataSystem，另一个是 BPTree。

DataSystem 类用来统筹地管理所有的资源控制，包括用于读写数据的

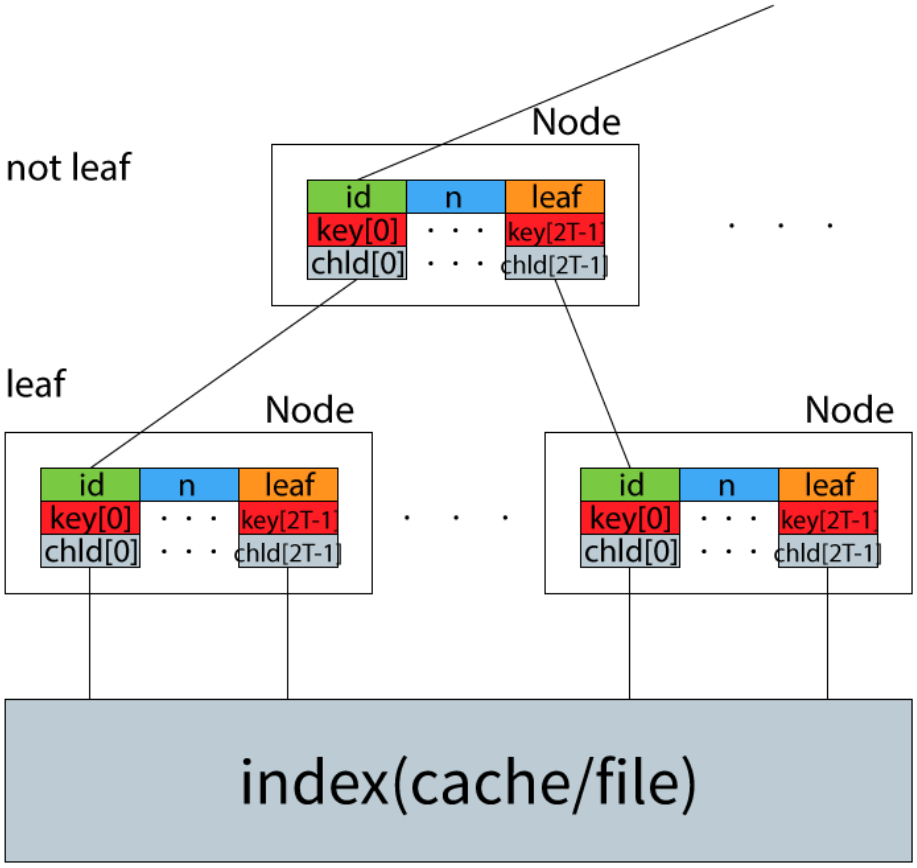
dataFile、用于标记空闲区块头的 idleHead 和用来实现数据索引 B+树的 indexTree。

dataFile 通过内存映射技术实现对于 data.txt 的快速读取和写入，通过 mmap 函数，将文件从物理存储器中映射至虚拟存储器，从而实现比直接文件操作更快速的读写操作。在这里，我并没有把整个 data.txt 全部进行了内存映射，而是对 data 进行了 4k 对齐的操作，使得每 4096 个 Byte 可以形成一页，没有跨页的数据。每次映射以页的单位进行，既节省了内存，又不浪费多少时间。

idleHead 则是通过链表的形式，把 dataFile 中的空闲区块有序记录下来，通过上文所述的 best fit 技术，少量牺牲时间的同时，换取更大的空间效率。

至于 indexTree 的具体细节，将在下一部分数据结构中具体介绍。

数据结构——B+树



如上文所述，一个 B+树的结点的结构包含：

$$\mathrm{id} + \mathrm{n} + \mathrm{leaf} + \mathrm{key}[0..2 * \mathrm{T} - 1] + \mathrm{child}[0..2 * \mathrm{T} - 1]$$

其中各项所代表的意义如下：

id - 记录该结点的 id，用于 cache 比对和索引位置的确定。

n - 记录有多少个子结点，如此可以在删除时不用擦除无用数据。

key[0..2 * T - 1] - 用来存储每个结点所代表的索引值，除了叶结点之外，每个结点都必须有 T~2 * T 个（B+树性质）

child[0..2 * T - 1] - 用来存储子结点在 index.txt 中的位置，与 B 树不同的地方，在于这颗 B+树的 key 值数量和 child 值数量相同，每个子结点所存储的结点 key 值范围在$\mathrm{key}[i] \leq \mathrm{childKeys} < \mathrm{key}[i+1]$，若该结点为最后一个 child 结点，则子结点所存储的结点 key 值范围在$\mathrm{key}[i] \leq \mathrm{childKeys} < \infty$。

leaf - 这是一个标记值，用来标记当前结点是否为叶子结点。若当前结点不是叶子结点，则表示 child 值指向的是自己的子结点；若当前结点是叶子结点，则表示 child 值指向的是 data.txt 的位置。

通过 B+树的插入、删除、分裂等操作，实现整个数据存储系统的各项操作（其中细节此处不做赘述）。

内存中并不全盘存储整颗 B+树，相反，常驻内存的仅有 root 一个结点，其他的结点都是 demand fetch，在需要时才从 index.txt 中读取。但是，一次插入和删除操作需要大量对于结点的读取，因此每次对于文件的读取和写入较大地浪费了时间。为了改进这方面的效率，我采用了一个包含 CACHESIZE 个 set 的 direct cache，相关的伪代码如下：

```
if (cache[child[x] / 4 % CACHESIZE].id == id)
   // cache hit
   do load or write work
else // cache miss
   write old index to file
   read new index from file
   do load or write work
```

这里要除以 4 的原因在于，作为一个 struct 是需要 4Byte 对齐的，因此如果单纯地对 CACHESIZE 取模，会造成 75%的浪费，因此除以 4 能够更大地提

高缓存的效用。

有了这样的缓存，只有 cache miss 的时候才需要对物理存储器进行读取，其他时候都能直接从内存中读取，减少了从硬盘读取所浪费的时钟数。

性能测试

测试一：插入测试

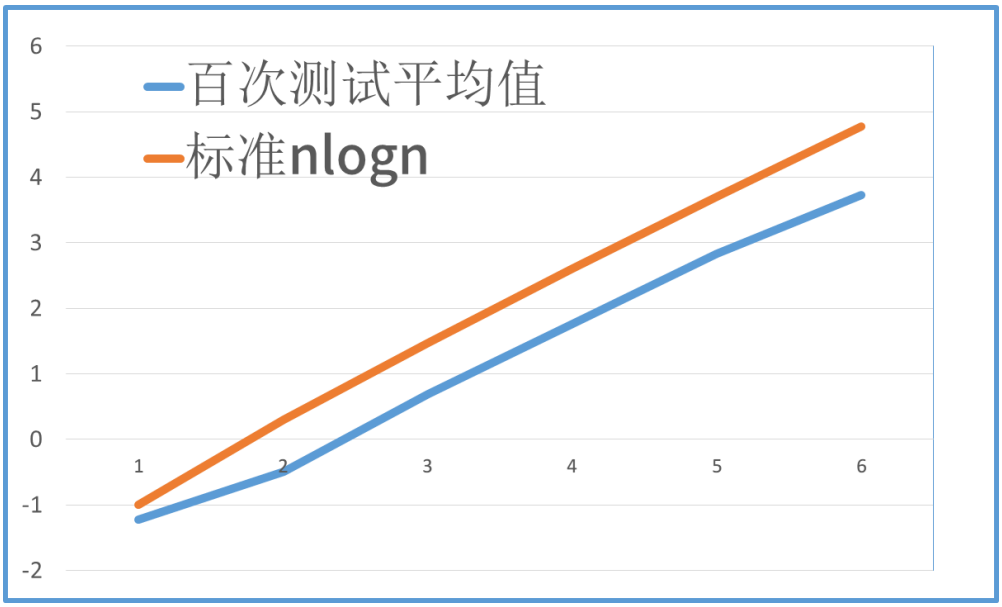
相关伪代码：

```
for (i = 1 to num)
     key = randomString
     value = randomString
     t1 = clock()
     dataSystem.insertItem(key, value)
     t2 = clock()
     totalTime += t2 - t1
```

测试结果：

num(nodes)	totalTime(ms)
10	0.05887
100	0.31836
1000	4.94513
10000	57.1916
100000	679.603
1000000	5408.89

数据经过取对数处理后和 nlogn 的趋势对比图:



结果分析:

从图中我们可以看到, 基本上和理论值 nlogn (单个插入需要 logn, 共 n 个) 的趋势非常接近, 因此整个 B+树的索引实现在插入部分是正常且正确的。

测试二: 查找测试

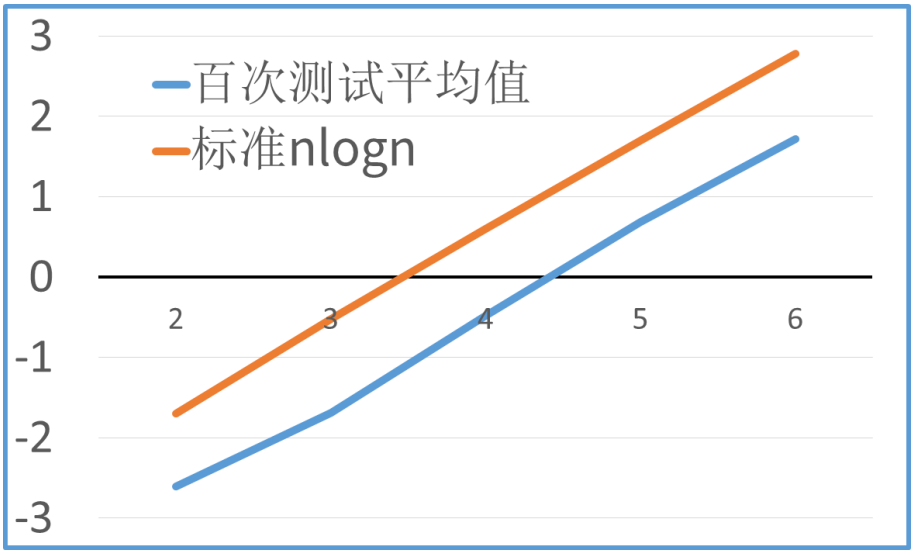
相关伪代码:

```
for (i = 1 to num)
     insertNodes
     data[num / 100] = getRandomNodes
for (j = 1 to num / 100)
     t1 = clock()
     dataSystem.getItem(data[j])
     t2 = clock()
     totalTime += t2 - t1
```

测试结果:

num(nodes)	totalTime(ms)
100	0.00246
1000	0.02068
10000	0.32719
100000	4.8443
1000000	52.1962

数据经过取对数处理后和 nlogn 的趋势对比图:



结果分析:

从图中我们可以看到, 基本上和理论值 nlogn (单个查找需要 logn, 共 n 个) 的趋势非常接近, 因此整个 B+树的索引实现在查找部分是正常且正确的。

测试三: 替换测试

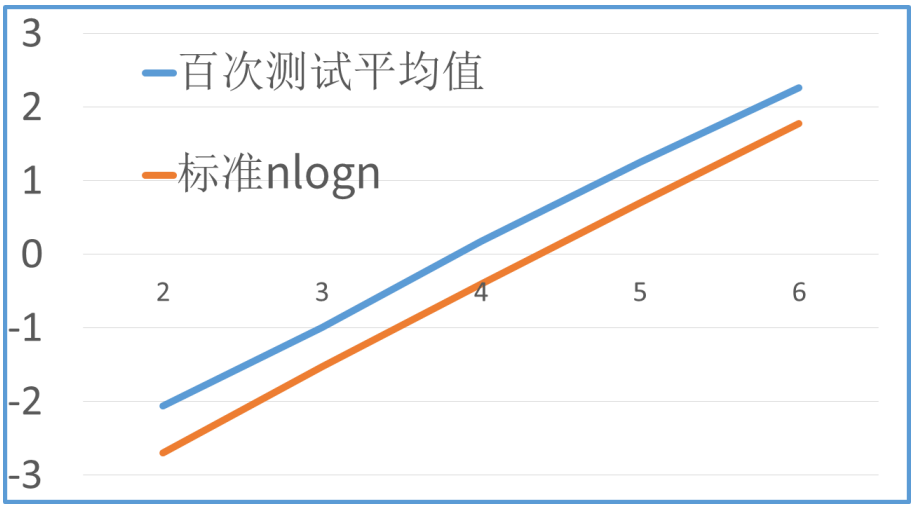
相关伪代码:

```
for (i = 1 to num)
     insertNodes
data[num / 100] = getRandomNodes
for (j = 1 to num / 100)
     t1 = clock()
     dataSystem.modifyItem(data[j])
     t2 = clock()
     totalTime += t2 - t1
```

测试结果:

num(nodes)	totalTime(ms)
100	0.0087
1000	0.10123
10000	1.49881
100000	17.955
1000000	182.797

数据经过取对数处理后和 nlogn 的趋势对比图:



结果分析:

从图中我们可以看到, 基本上和理论值 nlogn (单个替换需要 logn, 共 n 个) 的趋势非常接近, 因此整个 B+树的索引实现在替换部分是正常且正确的。

测试四: 删除测试

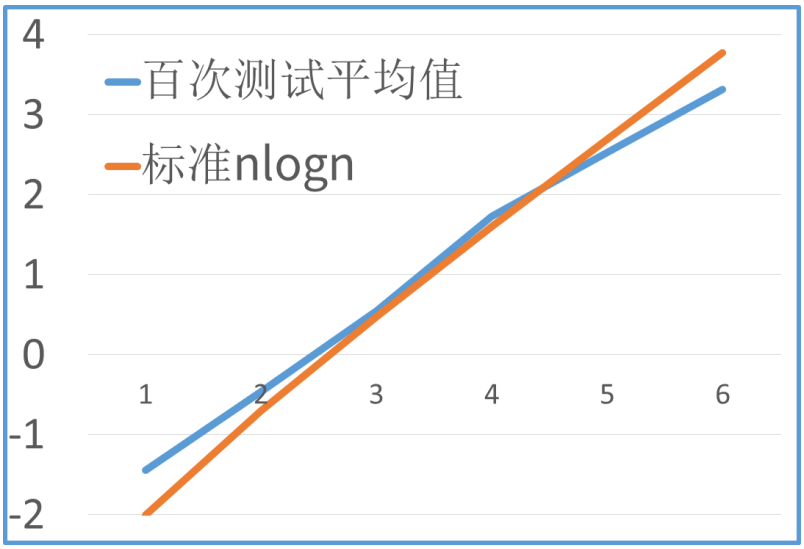
相关伪代码:

```
for (i = 1 to num)
     insertNodes
     data[i] = key
for (i = 1 to num)
     t1 = clock()
     dataSystem.deleteItem(data[i],newString)
     t2 = clock()
     totalTime += t2 - t1
```

测试数据:

num(nodes)	totalTime(ms)
10	0.03587
100	0.3497
1000	3.51689
10000	54.0853
100000	338.425
1000000	2061.72

数据经过取对数处理后和 nlogn 的趋势对比图:



结果分析:

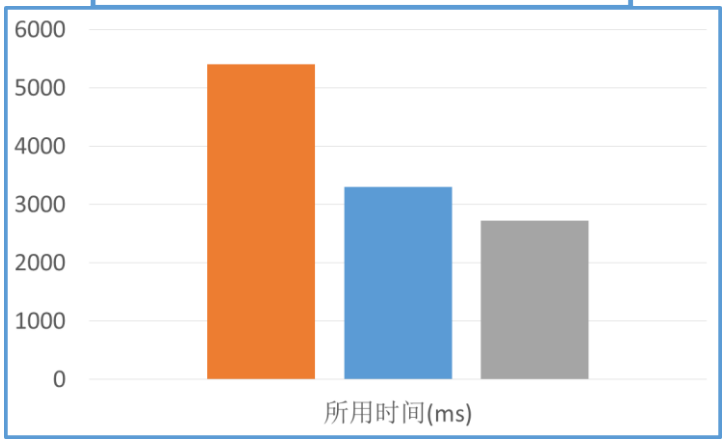
从图中我们可以看到, 基本上和理论值 nlogn (单个删除需要 logn, 共 n 个) 的趋势非常接近, 因此整个 B+树的索引实现在删除部分是正常且正确的。

测试五: CACHESIZE 测试

测试方法: 修改 index 的 cache 大小

测试结果:

缓存大小	所用时间(ms)
100	5408.89
1000	3303.57
3000	2725.1



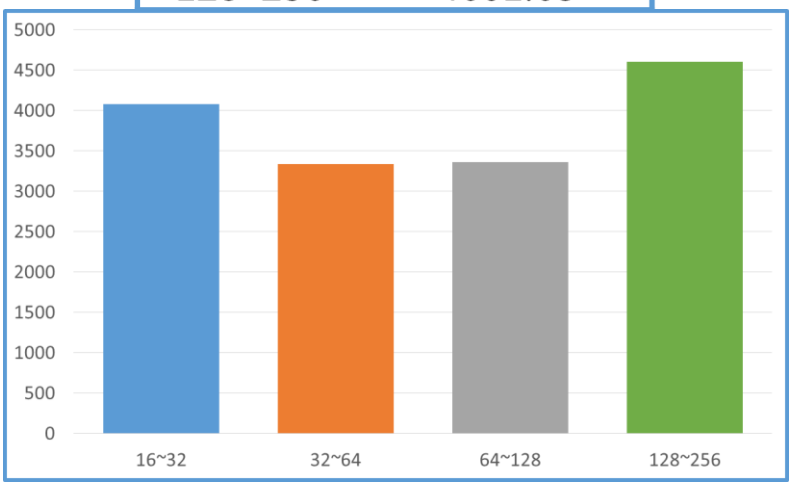
结果分析:

如理论所示, 通过增加 cache 的大小, 确实可以极大地优化所用的时间, 然

而更大的 cache 往往带来的是更大的内存占用，如我的程序就不能开出 5000 的 cache。因此需要在两者之间进行更好的权衡。

测试六：结点宽度测试

缓存宽度	所用时间(ms)
16~32	4080.11
32~64	3337.26
64~128	3363.57
128~256	4601.63



结果讨论：

就理论而言，减小结点宽度会增加 B+树的高度，增加结点从硬盘获取的次数从而增加时间，同时也会减小每个结点中线性比较的范围，从而节约时间；增加结点宽度的作用于此相反。因此，结点的宽度是需要好好斟酌的。就数据而言，结点宽确实有这样两个维度的影响，并且在 32~64 和 64~128 宽度情况下表现出色。因此测试结果符合理论预期。

仍需改进之处：

整个简易数据存储系统仍有一些需要改进的地方，我认为主要有以下两点：

第一、在 index.txt 中，为记录被删除结点所留下的空档，使得这些结点在被删除后留下的空闲区块无法被后来的结点使用，造成一定的空间浪费。可以像 data.txt 一样，记录空档位置，并择优插入。

第二、整个 idle 链表的排序是由线性的插入排序实现，在大数据下插入排序的时间复杂度较高。应采用堆排序的方法进行优化，提高整个系统的时间效率。