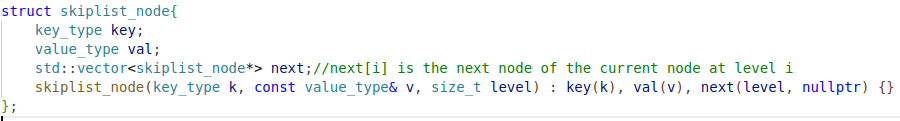
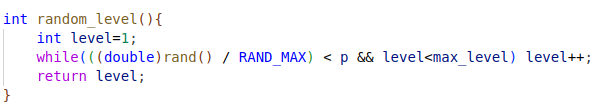
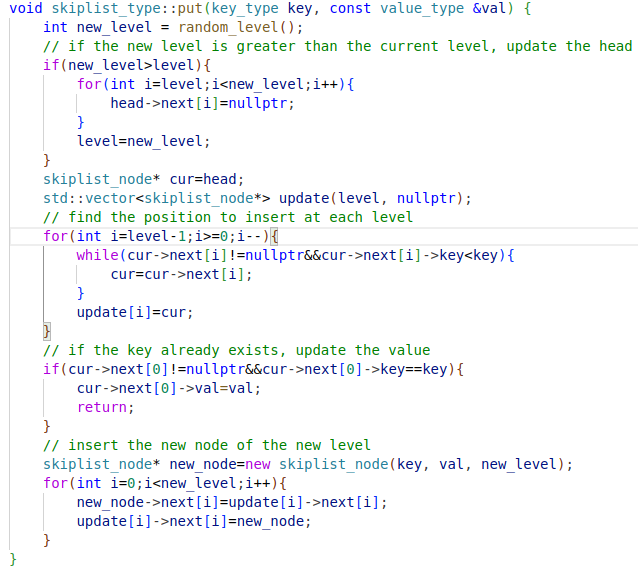
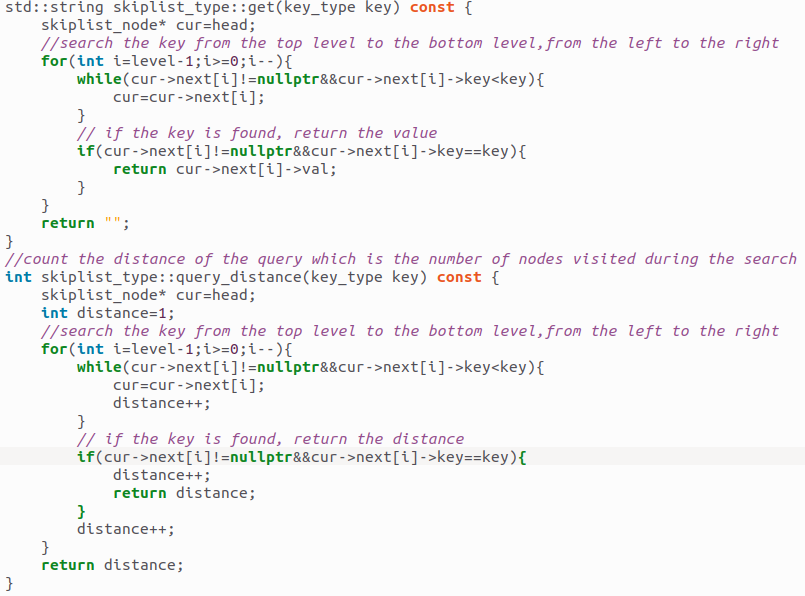
**HW1**

1. 关键代码说明：
2. 跳表节点：如下图。next结构是一个节点指针的vector 用于储存不同层数的后继节点，next[i] 就是第i层的节点。
3. ****层数生成：如下图。

****

1. 插入（更新）节点的put函数：如果随机出的层数大于当前最高层，就为head头节点new对应的指针。然后需要找到每一层的插入位置，即每一层的前继节点。遵循从上到下、从左到右的查询顺序进行遍历，把找到的节点维护在update数组里。最后插入新节点，若已经存在就更新，然后把各个层的前继节点以及后继节点与新节点相连。

****

4. 查询get函数：见上图。逻辑同上，从上到下从左到右遍历。不同的是，查询无需降到最底层，若已经查找到key，则直接返回。

5. 计算查询路径长度的query\_distance函数：见上图。同查询，只需要把return改成distance即可。因为是所有节点的个数，所以层数变化时也需要加1。

1. 实验结果见下图。

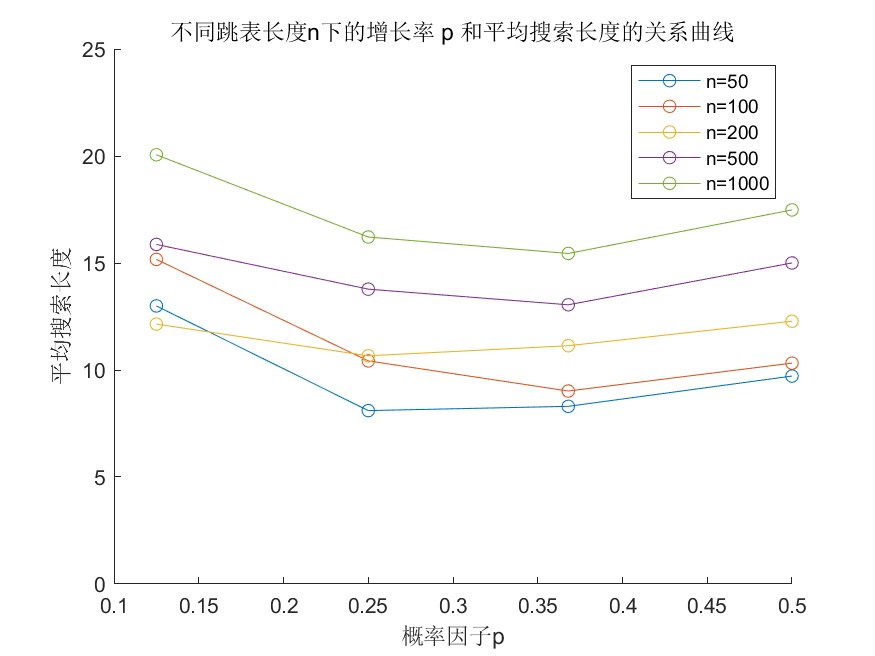
发现在n不断增大的情况下，p和平均搜索长度的关系曲线趋势大致相同，即在1/4到1/e之间有最小值，而在区间外平均搜索长度变大。这与课上所讲的分析结果是符合的（我们可以认为搜索时间和搜索长度近似成正比）。

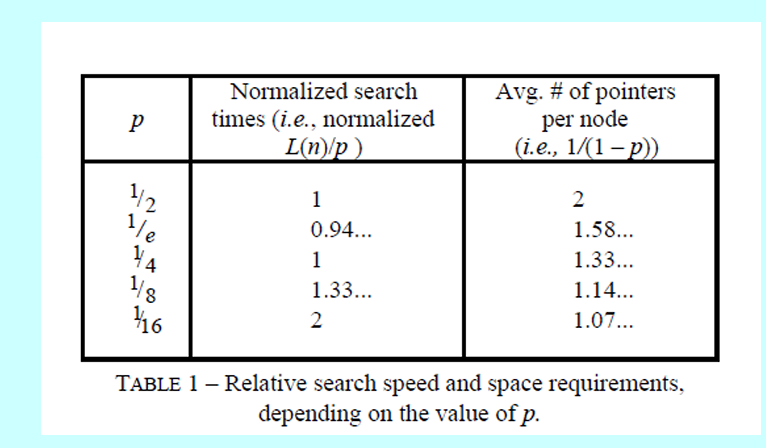
从理论上分析，可能有如下原因：

·p过小时，跳表的层数随之变小，意味着大部分搜索在低层进行，跳表“跳跃”的功能没有得到利用，增加了搜索路径的长度；

·p过大时，跳表的高层的节点个数也变多，意味着高层可能仍然像低层一样比较“完整”，也就失去了其“跳跃”的功能，增加了搜索路径的长度；

·p比较适中时，跳表的“跳跃”性能比较好，就如二分查找一样，在查询时能够进行合适的范围跳跃，也就优化了搜索长度和搜索时间。





**相关分析得到的不同p的搜索时间表**