HW1

一、关键代码说明：

1. 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。

```
struct skiplist_node{
    key_type key;
    value_type val;
    std::vector<skiplist_node*> next;//next[i] is the next node of the current node at level i
    skiplist_node(key_type k, const value_type& v, size_t level) : key(k), val(v), next(level, nullptr) {}
};
```

2. 层数生成：如下图。

```
int random_level(){
    int level=1;
    while(((double)rand() / RAND_MAX) < p && level<max_level) level++;
    return level;
}
```

3. 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。

```
void skiplist_type::put(key_type key, const value_type &val) {
    int new_level = random_level();
    // if the new level is greater than the current level, update the head
    if(new_level>level){
        for(int i=level;i<new_level;i++){
            head->next[i]=nullptr;
        }
        level=new_level;
    }
    skiplist_node* cur=head;
    std::vector<skiplist_node*> update(level, nullptr);
    // find the position to insert at each level
    for(int i=level-1;i>=0;i--){
        while(cur->next[i]!=nullptr&&cur->next[i]->key<key){
            cur=cur->next[i];
        }
        update[i]=cur;
    }
    // if the key already exists, update the value
    if(cur->next[0]!=nullptr&&cur->next[0]->key==key){
        cur->next[0]->val=val;
        return;
    }
    // insert the new node of the new level
    skiplist_node* new_node=new skiplist_node(key, val, new_level);
    for(int i=0;i<new_level;i++){
        new_node->next[i]=update[i]->next[i];
        update[i]->next[i]=new_node;
    }
}
```

```
std::string skiplist_type::get(key_type key) const {
    skiplist_node* cur=head;
    //search the key from the top level to the bottom level,from the left to the right
    for(int i=level-1;i>=0;i--){
        while(cur->next[i]!=nullptr&&cur->next[i]->key<key){
            cur=cur->next[i];
        }
        // if the key is found, return the value
        if(cur->next[i]!=nullptr&&cur->next[i]->key==key){
            return cur->next[i]->val;
        }
    }
    return "";
}
//count the distance of the query which is the number of nodes visited during the search
int skiplist_type::query_distance(key_type key) const {
    skiplist_node* cur=head;
    int distance=1;
    //search the key from the top level to the bottom level,from the left to the right
    for(int i=level-1;i>=0;i--){
        while(cur->next[i]!=nullptr&&cur->next[i]->key<key){
            cur=cur->next[i];
            distance++;
        }
        // if the key is found, return the distance
        if(cur->next[i]!=nullptr&&cur->next[i]->key==key){
            distance++;
            return distance;
        }
        distance++;
    }
    return distance;
}
```

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。

二、实验结果见下图。

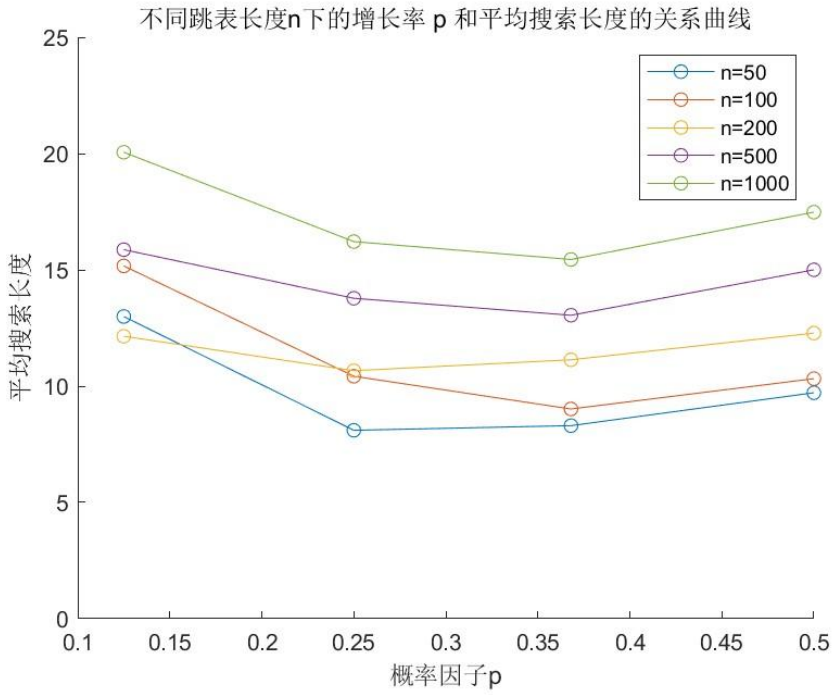
发现在 n 不断增大的情况下，p 和平均搜索长度的关系曲线趋势大致相同，即在 1/4 到 1/e 之间有最小值，而在区间外平均搜索长度变大。这与课上所讲的分析结果是符合的（我们可以认为搜索时间和搜索长度近似成正比）。

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。



p	Normalized search times (*i.e.*, normalized $L(n)/p$)	Avg. # of pointers per node (*i.e.*, $1/(1-p)$)
$1/2$	1	2
$1/e$	0.94...	1.58...
$1/4$	1	1.33...
$1/8$	1.33...	1.14...
$1/16$	2	1.07...

TABLE 1 – Relative search speed and space requirements, depending on the value of p.

相关分析得到的不同 p 的搜索时间表