Homework 1: Skiplist

赵楷越 522031910803

1 对跳表的关键代码的说明

1.1 搜索 get

```
66  std::string skiplist_type::get(key_type key) const
67  {
68      skiplistNode *x = header;
69      for (int i = currentLevel; i >= 1; i--)
70      {
71          while (x->forward[i] != nullptr && x->forward[i]->key < key)
72              x = x->forward[i];
73      }
74      x = x->forward[1];
75      if (x != nullptr && x->key == key)
76          return x->value;
77      else
78          return "";
79  }
```

在搜索 get 函数的实现中，采取了原论文中伪代码的经典算法。这里要注意，在 71 行和 75 行时要判断当前指针在 forward 的后继是否为空的情况，因为本 skiplist 类中并没有将最大侧的 tailer 引入，因此，若 x->forward[i] 指向 NIL 时，其为空指针。将该种情况纳入考虑后完成了搜索 get 函数的实现。

1.2 随机层数生成 randomLevel

```
8   int skiplist_type::randomLevel()
9   {
10      int level = 1;
11      while (static_cast<double>(std::rand()) / RAND_MAX < p && level <= maxLevel)
12      {
13          level++;
14      }
15      return level;
16  }
```

因为在本 skiplist 类中我们将最底层设置为 1 层，因此此处的 level 从 1 开始，根据生成的 0，1 之间的随机数与 p 的比较结果，模拟随机层数的生成。同时要保证 level 不超过 skiplist 的最大层数。

1.3 插入 put

```
35  void skiplist_type::put(key_type key, const value_type &val)
36  {
37      std::vector<skiplistNode *> update(maxLevel + 1, nullptr);
38      skiplistNode *x = header;
39      for (int i = currentLevel; i >= 1; i--)
40      {
41          while (x->forward[i] != nullptr && x->forward[i]->key < key)
42              x = x->forward[i];
43          update[i] = x;
44      }
45      x = x->forward[1];
46      if (x != nullptr && x->key == key)
47          x->value = val;
48      else
49      {
50          int level = randomLevel();
51          if (level > currentLevel)
52          {
53              for (int i = currentLevel + 1; i <= level; i++)
54                  update[i] = header;
55              currentLevel = std::min(level, maxLevel);
56          }
57          x = new skiplistNode(level, key, val);
58          for (int i = 1; i <= level; i++)
59          {
60              x->forward[i] = update[i]->forward[i];
61              update[i]->forward[i] = x;
62          }
63      }
64  }
```

采取了原论文中经典的算法设计。插入时在对查找值 key 的过程中写入待更新指针数组 update。如果找到了 key 值节点，不做多余处理。如果没找到 key 值节点，便新建一个 skiplist 节点插入其中，并根据 update 数组将新节点连接入原表中。

1.4 查找长度 querydistance

计算查找长度的函数与搜索的函数大体没有差异。但是要注意这里的查找长度逻辑与搜索逻辑略有不同，当查找到了目标节点之后，无需下降到最底层，查找立即结束。因此，引入了在第 89 行的 if 语句，用于判断是否已经找到了目标节点。同时，因为要将查找的初始 head 节点算入，因此 dis 从 1 开始计数。（代码见后一页）

```cpp
int skiplist_type::query_distance(key_type key) const
{
    int dis = 1;
    skiplistNode *x = header;
    for (int i = currentLevel; i >= 1; i--)
    {
        while (x->forward[i] != nullptr && x->forward[i]->key < key)
            x = x->forward[i], dis++;
        if(x->forward[i] != nullptr && x->forward[i]->key == key){
            dis++;
            return dis;
        }
        dis++;
    }
    x = x->forward[1];
    return dis;
}
```

2 作图及分析

我们选取了长度（跳表元素个数）分别为 50, 100, 200, 500, 1000，概率 p 分别为 1/2, 1/e, 1/4, 1/8 的跳表（共 20 对），每次测试时设置了不同的种子码，每次随机搜索次数设置为了 10000 次，并记录了每组对应的平均搜索长度，最终作出了表示相同跳表长度下，增长率 p 和平均搜索长度的关系的折线图如下。

在实验中，我们观察到在相同的跳表元素个数 Num 的情况下，随着增长率 p 的增加，平均搜索长度先减小后增加。同时，当 p 趋近于 0 时，平均搜索长度很大，当 p 在 1/e 附近时，平均搜索长度取到较小值。而在理论情况下，平均查找长度约等于 $(log_{1/p}n - 1)/p$，是一个随着 p 增加，先减后增的函数，其最小值在 1/e 附近，同时其随着跳表元素个数 n 的增加而增加。因此本次实验数据与理论情况相符合匹配。