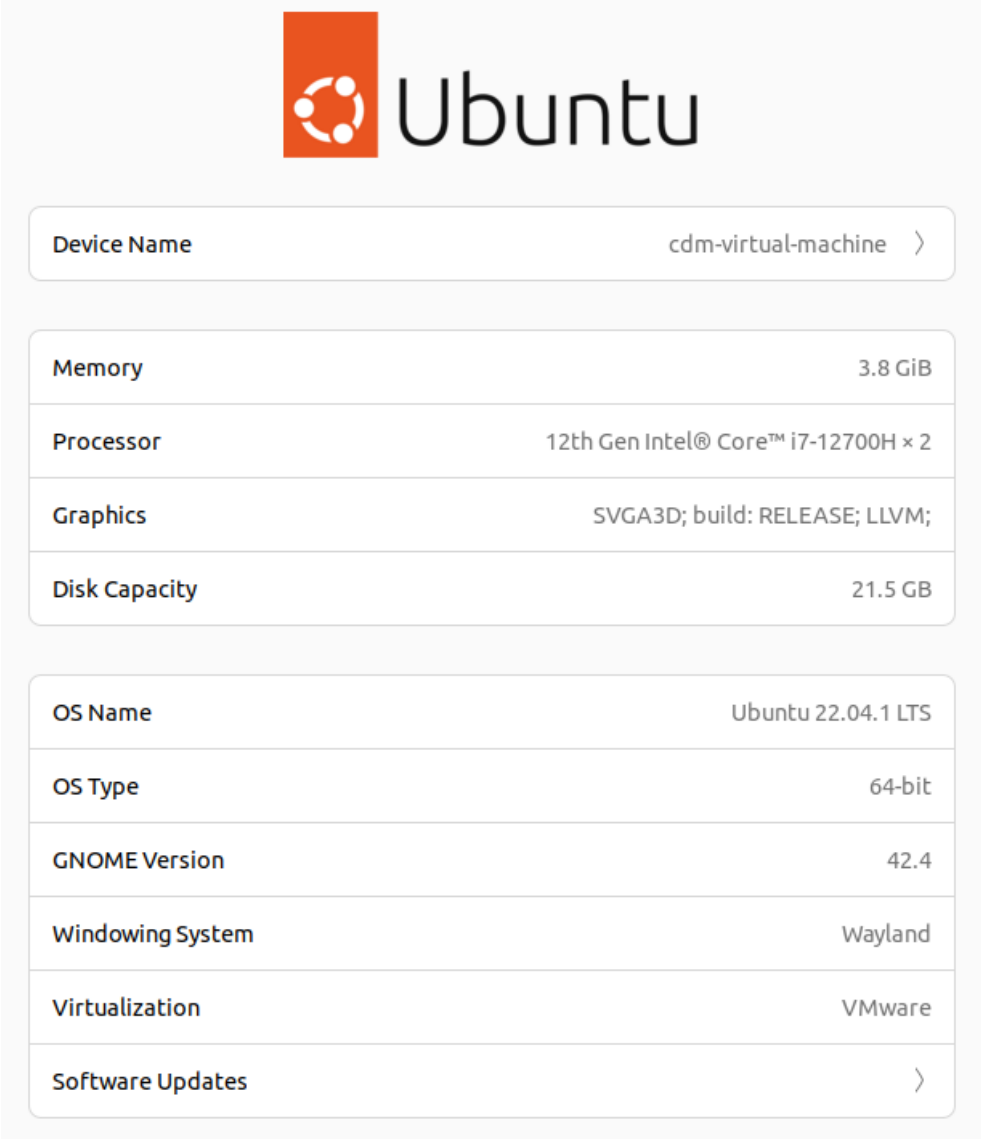
522031910747+李若彬+hw1

实验环境



Section 1 —— 跳表的实现

跳表类及节点结构体的实现

跳表类：

- 节点结构体
  - 所存的键值对（`key`，`val`）以及高度`level`
  - 一个指针指向其同层的下一个节点
  - 构造函数
  - 定义头指针和尾指针
- 构造函数
- 插入函数
- 查找函数
- 查找步长函数

- 概率
- 随机数（与概率用于生成节点高度）

```
class skiplist_type
{
        // add something here
private:
    struct Node
    {
        key_type key;
        value_type val;
        int level;
        Node *forward[MAX_LEVEL];
        Node(key_type key, const value_type &s, int level);
    } *head, *tail;
    int random_level();
    double my_rand();
    double Jump_Probaility;

public:
    explicit skiplist_type(double p = 0.5);
    void put(key_type key, const value_type &val);
    // std::optional<value_type> get(key_type key) const;
    std::string get(key_type key) const;

    // for hw1 only
    int query_distance(key_type key) const;
};
```

跳表插入的实现

- 首先要有一个指针`cur`，初始时指向最顶层的头指针，后续就使用这个指针由上至下、由左至右移动进行查找
- 还要有一个指针`update`用于存放上一次向下跳跃的节点后续会使用这个指针来更新跳表
- 具体算法：
  - `cur`指针首先从跳表"左上角"开始移动，比较当前节点的同层右侧节点的值和待插入的值，若右侧节点值更大或其为尾节点，则跳入下一层并用`update`记录下当前跳跃的节点，否则将当前的`cur`指针变更到此右侧节点
  - 当跳跃到最底层时，如果此时`cur`指针的节点不为尾节点或其值与待插入的值相同，则更新其`val`，如果此时`cur`指针的节点不为尾节点或其值不与待插入的节点值相同，则创建一个`n`节点并用`random_level()`初始化其高度，然后将其在概率生成的高度右侧指向`update`指针的节点原本的右侧节点，随后将当前`update`指针的节点指向这个插入的`n`节点完成插入操作

```
void skiplist_type::put(key_type key, const value_type &val){
    Node *cur = head;
    Node *updated[MAX_LEVEL];
    for (int i = MAX_LEVEL - 1; i >= 0; --i)
    {
        while (cur->forward[i] != NULL && cur->forward[i]->key < key)
            cur = cur->forward[i];
        updated[i] = cur;
```

```
    }
    if (cur->forward[0] != tail && cur->forward[0]->key == key)
    {
        cur->forward[0]->val = val;
        return;
    }
    int rlevel = random_level();
    Node *n = new Node(key, val, rlevel);
    for (int i = 0; i < rlevel; ++i)
        n->forward[i] = tail;
    for (int i = rlevel; i >= 0; --i)
    {
        n->forward[i] = updated[i]->forward[i];
        updated[i]->forward[i] = n;
    }
}
```

跳表查找的实现

- 类似于插入造作，首先要有一个指针`cur`，初始时指向最顶层的头指针，后续就使用这个指针由上至下、由左至右移动进行查找
- 比较`cur`右侧的节点值是否与待查找的值相同，若是则返回其`val`，否则进行比较，当当前层`cur`的节点右侧节点不为空且其值小于所查找的值时，`cur`向右移动，否则向下移动一层继续进行比较直至最后一层
- 若在最后一层仍未找到则返回一个空字符串

```
std::string skiplist_type::get(key_type key) const {
    Node *cur = head;
    for (int i = MAX_LEVEL - 1; i >= 0; --i){
    while (cur->forward[i] != tail && cur->forward[i]->key < key)
        cur = cur->forward[i];
    if (cur->forward[i] != tail && cur->forward[i]->key == key)
        return cur->forward[i]->val;
    }
    return "";
}
```

其余部分代码

```
//跳表构造函数
skiplist_type::skiplist_type(double p) {
    Jump_Probaility = p;
    head = new Node(0, "", MAX_LEVEL);
    tail = new Node(0, "", MAX_LEVEL);
    for (int i = 0; i < MAX_LEVEL; ++i) {
        head->forward[i] = tail;
        tail->forward[i] = tail;
    }
}
//查找路径（类似于查找）
int skiplist_type::query_distance(key_type key) const {
    Node *cur = head;
```

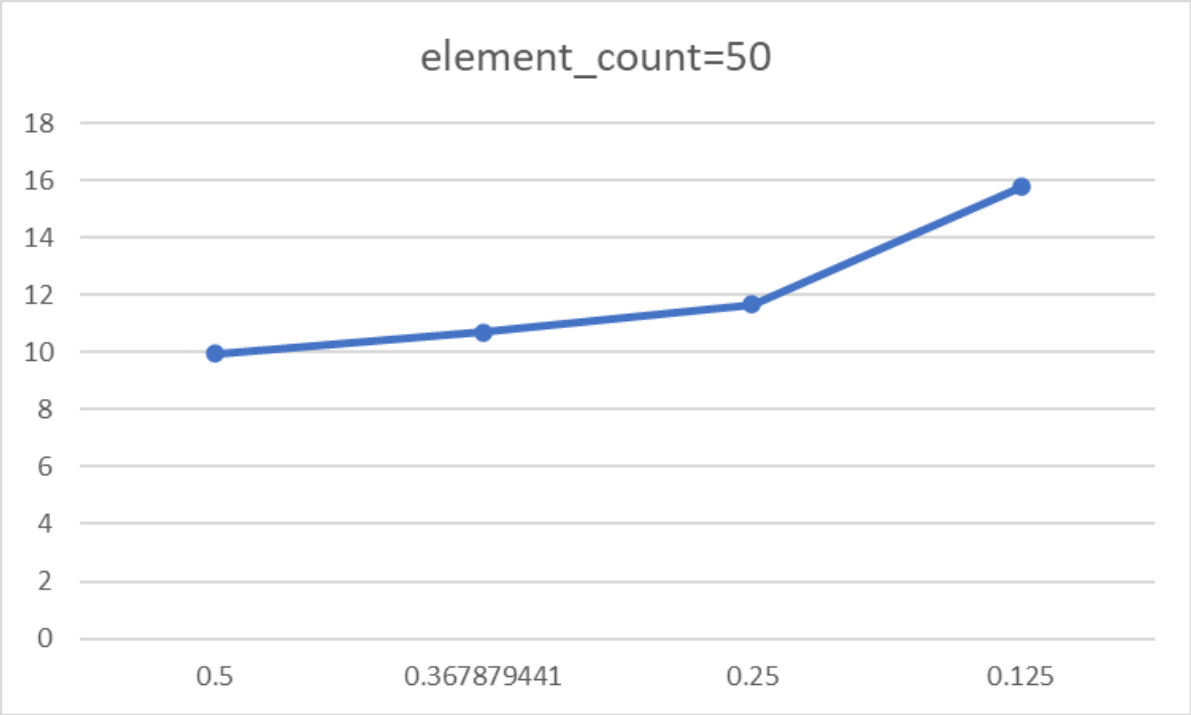
```
    int path_length = 1;
    for (int i = MAX_LEVEL - 1; i >= 0; --i){
        while (cur->forward[i] != tail && cur->forward[i]->key < key) {
            cur = cur->forward[i];
            path_length++;
        }
        if (cur->forward[i] != tail && cur->forward[i]->key == key) {
            return path_length++;
        }
        path_length++;
    }
    return path_length++;
}
```

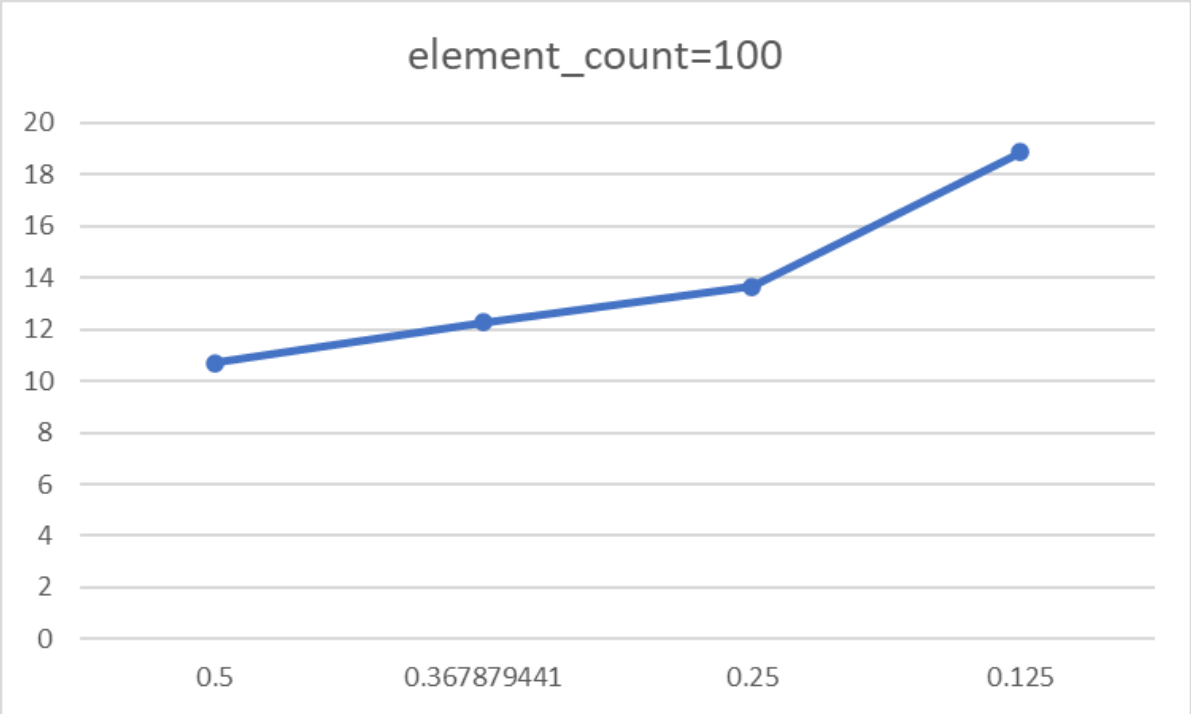
Section 2 —— 探究增长率对跳表性能的影响

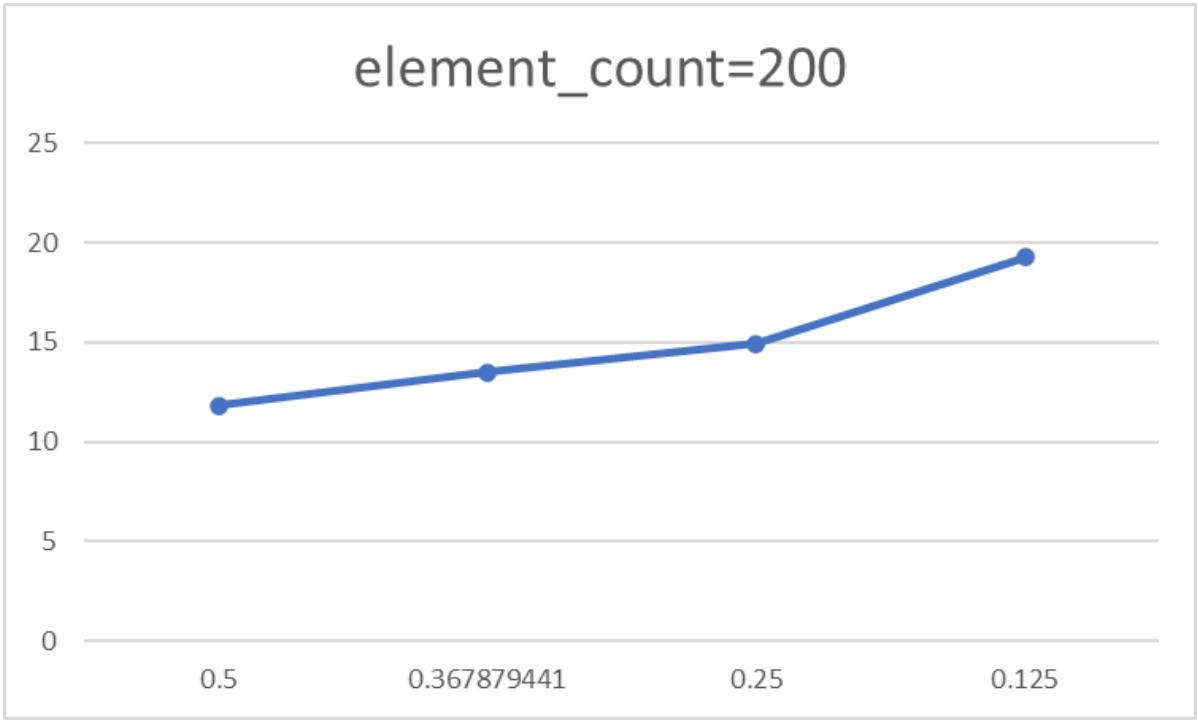
选择5个`element_count`即跳表元素个数$(50，100，200，500，1000)$，4个`p`值即概率$(1/2，1/e，1/4，1/8)$，分别测试搜索10000次的`average query distance`即平均搜索长度，结果如下

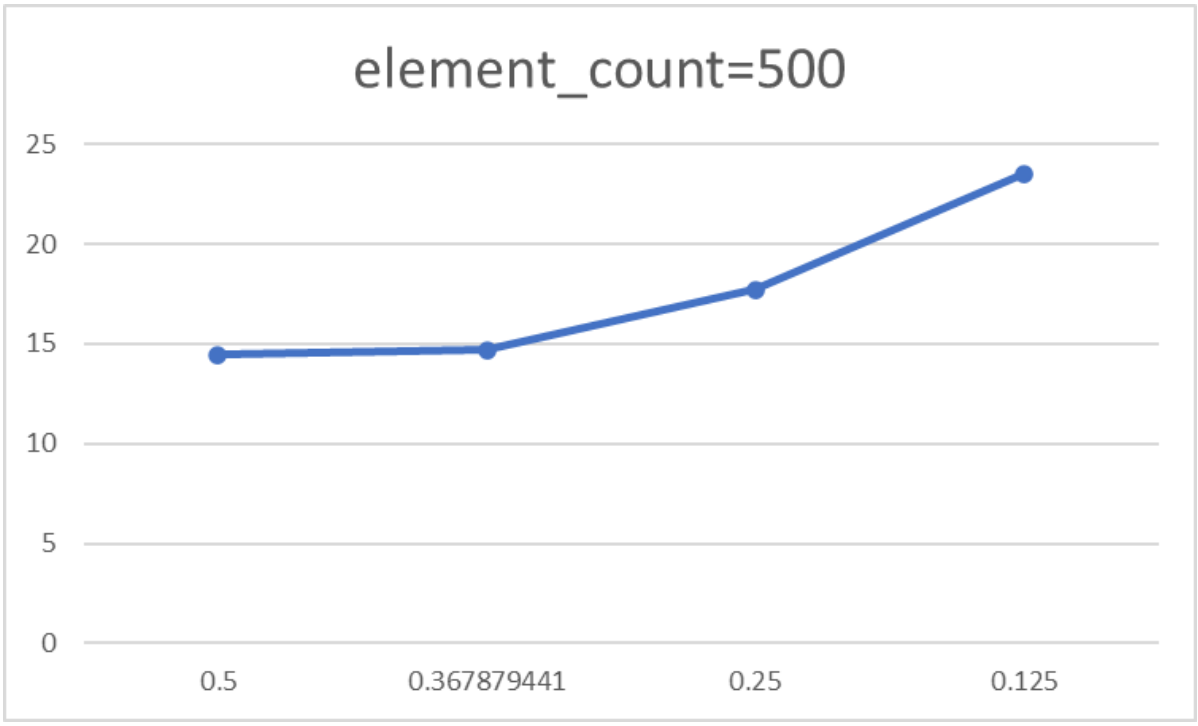
element_count	p	average query distance
50	0.5	9.93635028
50	0.367879441	10.68415525
50	0.25	11.67262122
50	0.125	15.76735273
100	0.5	10.70715588
100	0.367879441	12.26985
100	0.25	13.64880171
100	0.125	18.87082646
200	0.5	11.80721189
200	0.367879441	13.47543519
200	0.25	14.92413028
200	0.125	19.2740173
500	0.5	14.45684688
500	0.367879441	14.69501132
500	0.25	17.71731112
500	0.125	23.54042721
1000	0.5	18.87121512

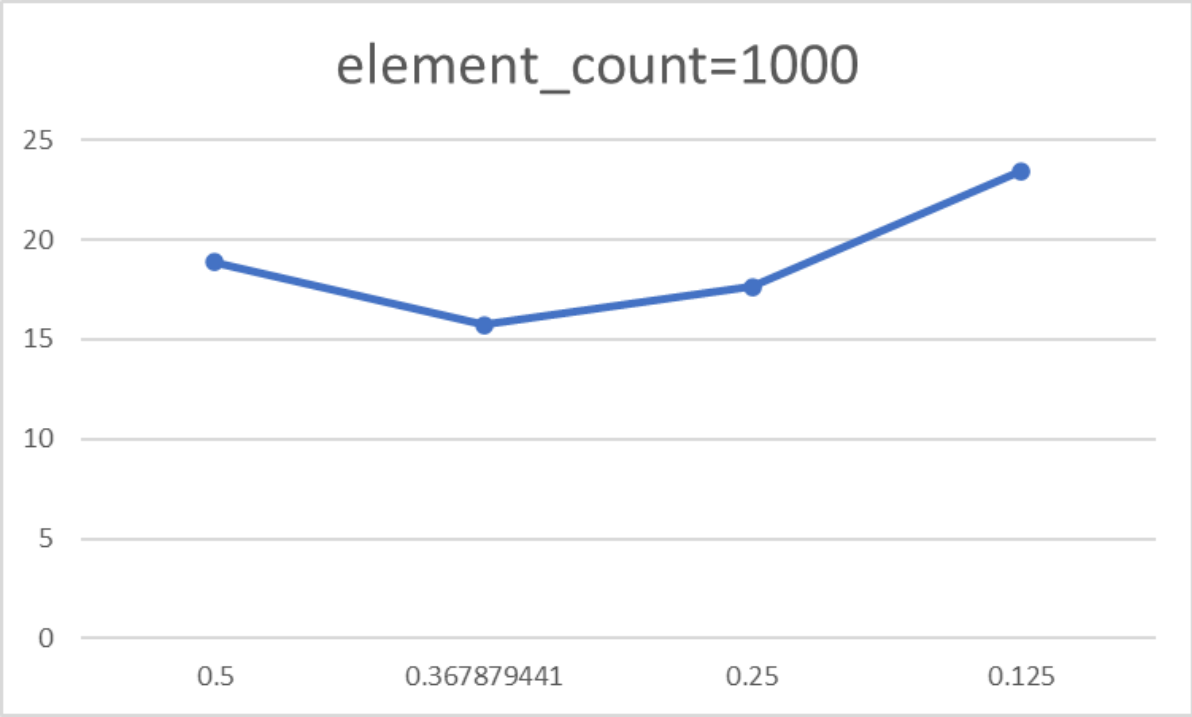
element_count	p	average query distance
1000	0.367879441	15.72349454
1000	0.25	17.64183862
1000	0.125	23.45992068






element_count=200
25
20
15
10
5
0
0.5
0.367879441
0.25
0.125


element_count=500
25
20
15
10
5
0
0.5
0.367879441
0.25
0.125



分析

由上述折线图可知：

- 首先，在`p`一定的前提下，随着`element_count`的增大，`average query distance`逐渐增加，因为整个链表变长了，搜索的长度会增加
- 然后，在`element_count`一定的前提下，随着`p`的减小，`average query distance`总体上呈现出先减后增的趋势，并且会在$1/e$处取得极值：
  - $L(n) = log_{1/p}n$，代表`MAX_LEVEL`，搜索的时间复杂度$C(i) = i/p$，$C(p) = \frac{ln(n)}{-pln(p)}$。对 $H(x) = xln(x)$ 求导：$H^{'}(x) = ln(x) + 1$，在$(0, \frac{1}{e}]$单调递减，$[\frac{1}{e}, +\infty]$单调递增，因此会在$1/e$处取得极值
- 在`element_count`较小时（例如50，100，200，500时），可能会出现偶然状况，比如查找的节点都大量的的集中在表头部分，此时结论并不严格成立，但是随`element_count`增大（例如1000时），结论会越来越明显