Hw1-Skiplist

1. Code Review

1.1 overview

In the implementation of **skiplist_type::node**, I utilize a vector composed of node pointers, labeled as "forward", to maintain the subsequent nodes' connections. Moreover, during the initialization process of the skiplist, the header is exclusively initialized, while **using nullptr to signify the tail NIL.**

```
class skiplist_type
{
struct node{
    key_type key;
    value_type val;
    std::vector<node *> forward; // Forward pointers
    node(key_type key,value_type value,int level);
};
public:
    node *header;
    int level;
    double p;
    //function...
};
```

1.2 Get

The get function of the skip list uses the curr pointer to locate the search results. It continuously probes forward at each level **until it encounters a value larger than the target, and then naturally drops to the next level.** This operation is implemented using a while loop. This process repeats until it stops at Level[0], at which point, the largest value smaller than the target is found. **The next node is the most likely location of the target value we are looking for.** At this point, compare the keys of the two to see if they are equal.

```
std::string skiplist_type::get(key_type key) const {
    node *curr = header;
    for( int i = level - 1;  i >= 0 ; i--){ // Traverse down the levels
       // If current node's key less than input, go forward
        while( curr->forward[i] && curr->forward[i]->key < key){
            curr = curr -> forward[i];
            }
```

```
            }
    curr = curr->forward[0];// Move to node with matching key
    if( curr && curr->key == key ){
        return curr->val;
        }
    else{
        return "";
    }
}
```

1.3 Put

Compared to the search function, the put function needs to **remember the reasonable position of the new node at each level**. Here, the update function is used to **record every drop node at each level**, which is the node before the new node. After the operation similar to the search is terminated, it is first **checked whether a node with the corresponding key already exists**. If it exists, just modify the val. Otherwise, a new node is added, which can be implemented using the method of adding elements to the linked list.

```
void skiplist_type::put(key_type key, const value_type &val) {
    // Tracks nodes to be updated after insert
    std::vector<node *> update(MAX_LEVEL, header);
    node *curr = header;
    for( int i = level - 1;  i >= 0 ; i--){
        while( curr->forward[i] && curr->forward[i]->key < key){
            curr = curr -> forward[i];
            }
         update[i] = curr;// Update tracking list
         }
    curr = curr->forward[0];
    if( curr && curr->key == key ){// If key exists, simply update its value
        curr->val = val;
        }
    else {// If new key, create new node
        int lv = randomLevel();
        level = std::max(level, lv);
        node *newNode = new node( key, val, level);
        for(int i = 0; i < lv; i++){
            newNode -> forward[i] = update[i]->forward[i];
            update[i] -> forward[i] = newNode;
            }
        }
    }
```

1.4 Other functions

```
// Node constructor initializing key, value and forward pointers
skiplist_type::node::node(key_type key,value_type value, int max_level = MAX_LEVEL):key(key),val(value),forward(max_level,nullptr){
}

// Skiplist constructor initializing header node, level and probability threshold
skiplist_type::skiplist_type(double p):dis(0,1) {
    this->header = new node(-1, "");
    this->level = 0;
    this-> p = p;
}

// Function to generate a random level for a new node
int skiplist_type::randomLevel(){
    int lv = 1;
    while( lv < MAX_LEVEL && dis(gen) < p){
        lv++;
    }
    return lv;
}

// Function to query the distance (number of steps) to a specific key
int skiplist_type::query_distance(key_type key) const {
    int step = 0;
    node *curr = header;
    for( int i = level - 1;  i >= 0 ; i--){
        step++;
        while( curr->forward[i] && curr->forward[i]->key < key){
            curr = curr -> forward[i];
                step++;
            }
           if(curr->forward[i]->key == key){ //terminated if found by advance
                step++;
               return step;
            }
         }
         step++;
    return step;
}
```

2. Experimental Results and Analysis

2.1 Experimental Results

Select **5 Element_count** $(50, 100, 200, 500, 1000)$ **, and 4 P_value values** （ $1/2, 1/e, 1/4, 1/8)$ ， respectively test the average number of steps of Search 10000 times each and the results are as follows

Element_count	**P_Value**	**Average_Query_Distance**	**Expected_Query_Distance**
50	0.5	10.9712	13.2877
50	0.367879441	9.4332	12.2160
50	0.25	8.8755	12.6210
50	0.125	9.3889	16.1931
Element_count	**P_Value**	**Average_Query_Distance**	**Expected_Query_Distance**
100	0.5	12.4932	15.2877
100	0.367879441	10.6989	14.1001
100	0.25	10.0885	14.6210
100	0.125	11.0284	18.8598
Element_count	**P_Value**	**Average_Query_Distance**	**Expected_Query_Distance**
200	0.5	13.9769	17.2877
200	0.367879441	11.9874	15.9843
200	0.25	11.3487	16.6210
200	0.125	12.4901	21.5265
Element_count	**P_Value**	**Average_Query_Distance**	**Expected_Query_Distance**
500	0.5	15.9346	19.9316
500	0.367879441	13.7269	18.4750
500	0.25	13.0336	19.2649
500	0.125	14.3950	25.0516
Element_count	**P_Value**	**Average_Query_Distance**	**Expected_Query_Distance**

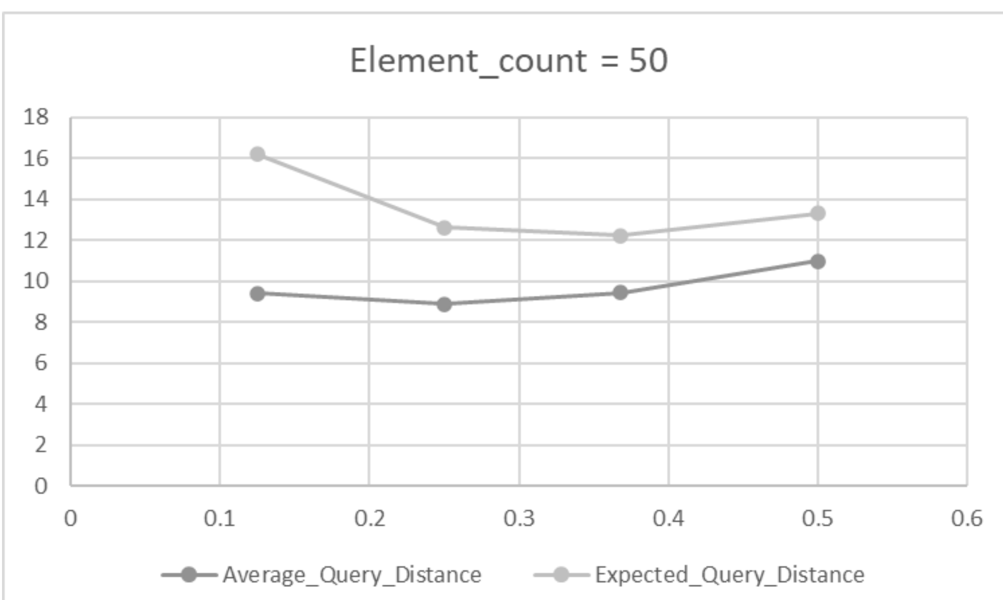
Element_count	P_Value	Average_Query_Distance	Expected_Query_Distance
1000	0.5	17.4554	21.9316
1000	0.367879441	14.9961	20.3592
1000	0.25	14.2227	21.2649
1000	0.125	15.8813	27.7183

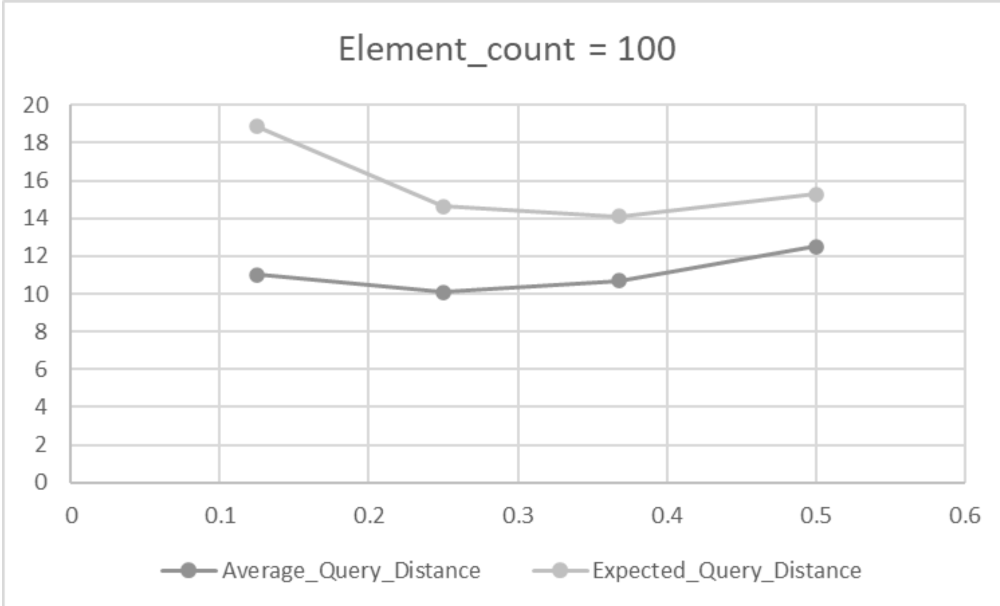
2.2 Analysis

By comparing and analyzing the experimental results with the theoretical $\frac{\log_{\frac{1}{p}} N}{p} + \frac{1}{1-p}$ upper bound , **it can be seen that the larger the value of p, the closer the average step number comes to the theoretical upper bound. The smaller the p value, the larger the difference from the theoretical upper bound, and this is more apparent when n is relatively small.**

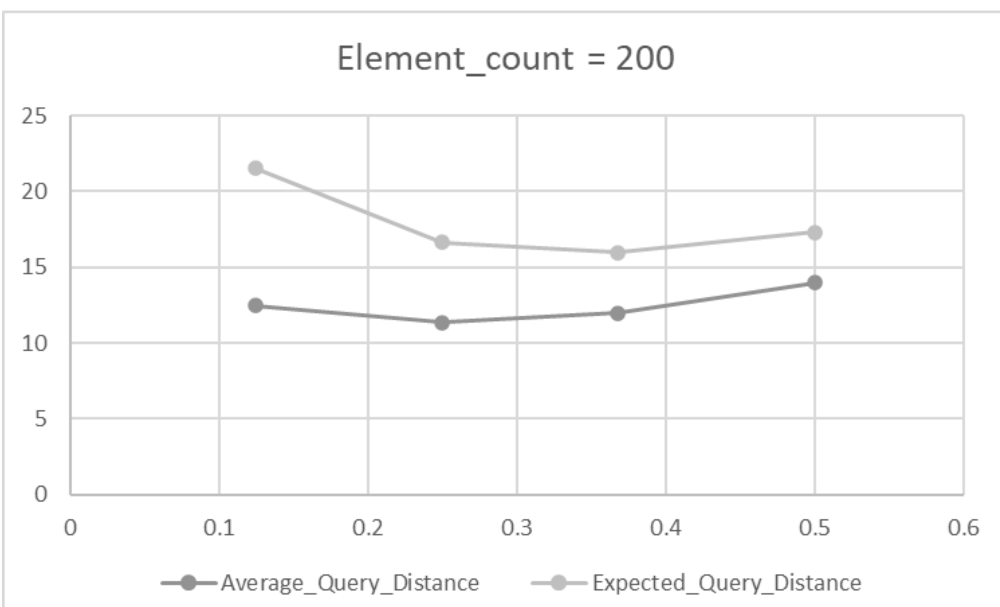
To explain this in a way easy to understand is that when the p-value is larger, **a greater number of nodes can ascend to higher levels, which enables the overall list structure to rapidly traverse across numerous nodes when performing a search operation**, thereby reducing the average step count for search operations. Considering the theoretical upper limit formula , an increase in the p-value leads to a decrease in the theoretical maximum number of steps for a search procedure.

In mathematical view, when p, n are small, The error of this upper bound estimation $1-(1-p^k)^n < np^k$ is quite substantial.




Element_count = 100
20
18
16
14
12
10
8
6
4
2
0
0
0.1
0.2
0.3
0.4
0.5
0.6
Average_Query_Distance
Expected_Query_Distance


Element_count = 200
25
20
15
10
5
0
0
0.1
0.2
0.3
0.4
0.5
0.6
Average_Query_Distance
Expected_Query_Distance


Element_count = 200
25
20
15
10
5
0
0
0.1
0.2
0.3
0.4
0.5
0.6
Average_Query_Distance
Expected_Query_Distance

3 Script

```
#Environment: Ubuntu22.04
import subprocess
import pandas as pd
import random
elements = [50, 100, 200, 500, 1000]
p_values = [1/2, 1/2.71828182846, 1/4, 1/8]
df = pd.DataFrame(columns=['Element_count', 'P_Value', 'Average_Query_Distance'])
for element in elements:
    for p_value in p_values:
        total_query_distance = 0
        for i in range(10000):
            seed = random.randint(0,100000000)
            command = f'./test-main {element} {seed} {p_value}'
            output = subprocess.check_output(command, shell=True).decode('utf-8')
            avg_query_distance = float(output.split('=')[-1].strip())
            total_query_distance += avg_query_distance

        avg_query_distance = total_query_distance / 10000

        new_row = pd.DataFrame({'Element_count': [element],
                                'P_Value': [p_value],
                                'Average_Query_Distance': [avg_query_distance]})
        df = pd.concat([df, new_row], ignore_index=True)

df.to_csv('result.csv', index=False)
```