Advanced Data Structure
Project IIX

Skip List

Individual Bonus: 洪奕迅

2024- 12- 21

Abstract

Skip List is a kind of ordered linked lists, and have a time complexity of $\mathcal{O}(logN)$ in deletion, insertion and searching. In this project, I implement skip list and test its practical performance in different datasets.

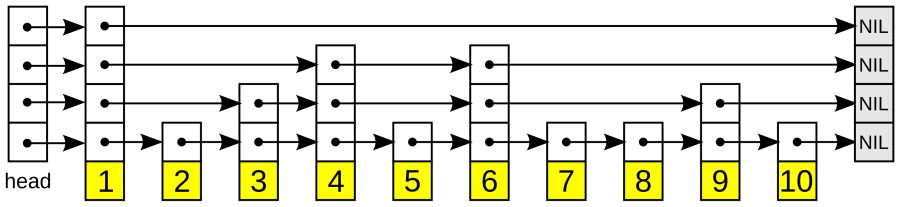
Index

1 Introduction

A skip list is a probabilistic data structure that allows $\mathcal{O}(\log n)$ average complexity for search as well as $\mathcal{O}(\log n)$ average complexity for insertion within an ordered sequence of n elements. The main characteristics of this structure is its probabilitic construction, leveled linked lists and connecting different levels.

2 Structure Illustration

2.1 Structure



Graph 1: Skip List Structure

This is a typical example of a skip list. The most significant feature of a skip list is that it's built in layers. The bottom layer contains all data and is a common linked lists. And each higher layer offers an access for searching elements in lower layer, which is really like B+ tree. And for a element in $layer_i$, it appears in $layer_{i+1}$ in a constant probability p (we define the layer order ascending from the bottom). So for a skip list, it totally contains $log_{\frac{1}{p}}N$ linked lists and the element in highest layer appears in every layer.

2.1.1 Operations

The excellent complexity of skip list depends on its layer feature, so the core of all operations is to maintain its layers correct and its ordered property.

2.1.2 Searching

To search an element in a skip list is quite easy. Thinking about searching in a B+ tree, we do that the same way. Starting from the highest layer, we compare each element in the layer, if element is found, since elements stored in each layer is the same vertically, we find the element. If not, we found the last element smaller than the element we want (if all linked lists are ascending) and go to next layer to do the scanning work. After scanning all layer, if we don't find the element, then it represents the elements wasn't in the skip list.

Complexity of searching is clear. Thinking we cannot find the element x in $layer_i$, and the last element smaller than x and the first element bigger than x is y and z, then we need at most scanning x, y and ω in $layer_{i+1}$ which qualifies $x \leq \omega \leq y$. So in each layer(except the highest and the second), we only need to scan over 3 elements, and in worst case, we scan from top to the bottom, bringing a complexity of $\mathcal{O}(3H)$ which is $\mathcal{O}(logN)$ totally.

2.1.3 Insertion

A simple idea is we use the searching operation and then insert the element in the position we found (after the last element smaller than it). But a trivial shortcoming is that if we don't update the upper layer, then the skip list will degenerate into a linked lists, so we need to update upper layers when insertion.

The easiest solution is we rebuild all upper layers after each insertion, which cause the complexity to become $\mathcal{O}(N)$, clearly something we don't want. So we introduce the probability in last section. We use a function called `randomLevel()` to do that:

```
int SkipList::randomLevel() {
    int level = 0;
    while (dist(gen) < probability && level < maxLevel - 1) {
        level++;
    }
    return level;
}
```

The function would indicate whether we need to insert the element in an exact upper layer by using random solutions. That insertion takes $C \cdot \mathcal{O}(1)$ and the total path are alike searching, so the time complexity is $\mathcal{O}(logN)$, too.

2.1.4 Deletion

The deletion operation are really alike the searching operation. But we need to delete the element in every layer if we find it. Each layer is a linked list, the deletion takes $\mathcal{O}(1)$, so the total complexity is the same as the searching operation - $\mathcal{O}(logN)$.

3 Implementation

3.1 Node Structure

For every node in the skip list, we design a node class containing value and a pointer to the next element:

```
class Node {
public:
    int value;
    std::vector<Node*> forward;

    Node(int value, int level);
};
```

And for the skip list, we design a skip list class for its layered architecture. Every node in it has its own level, probability and other values:

```
class SkipList {
private:
    Node* header;
    int maxLevel;
```

```
    float probability;
    int currentLevel;

    std::random_device rd;
    std::mt19937 gen; // For more randomly generated number
    std::uniform_real_distribution<float> dist;
```

3.2 Searching

The searching operation are detailed illustrated in the previous section, and in our structure, we offer all pointers we need. We just scan the whole list from top to the bottom and return whether the element is in the list:

```
bool SkipList::search(int value) {
    Node* current = header;

    // Form top to the bottom
    for (int i = currentLevel; i >= 0; --i) {
        while (current->forward[i] != nullptr && current->forward[i]->value < value) {
            current = current->forward[i];
        }
    }

    current = current->forward[0];  /
    return current != nullptr && current->value == value; // Return found or not
}
```

3.3 Insertion

In insertion, we firstly find the position to insert and then update all the paths down by randomly generate its layer using the `randomLevel()` function we've mentioned.

```
void SkipList::insert(int value) {
    std::vector<Node*> update(maxLevel + 1, nullptr);  // An array to record predecessors
    Node* current = header;

    // Find position for insertion from the top
    for (int i = currentLevel; i >= 0; --i) {
        while (current->forward[i] != nullptr && current->forward[i]->value < value) {
            current = current->forward[i];
        }
        update[i] = current;
    }
    current = current->forward[0];
    // Insertion
    if (current == nullptr || current->value != value) {
        int newLevel = randomLevel();  // Decide its position by random
        if (newLevel > currentLevel) {  // Update upper layers
            for (int i = currentLevel + 1; i <= newLevel; ++i) {
```

```
                update[i] = header;
            }
            currentLevel = newLevel;
        }
        Node* newNode = new Node(value, newLevel);
        for (int i = 0; i <= newLevel; ++i) {
            newNode->forward[i] = update[i]->forward[i];
            update[i]->forward[i] = newNode;
        }
    }
}
```

3.4 Deletion

Just the same as searching except deleting the elments all way down until searched in the bottom:

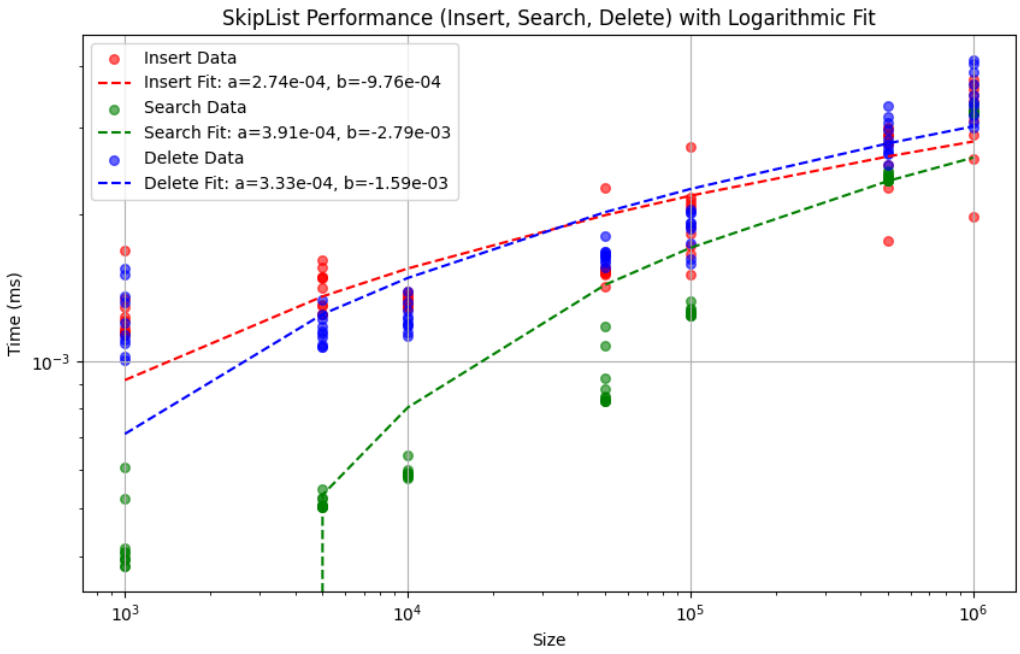
```
void SkipList::deleteValue(int value) {
    std::vector<Node*> update(maxLevel + 1, nullptr);
    Node* current = header;
    /* The same as searching, so skip it*/
    // Delete the element all way down
    if (current != nullptr && current->value == value) {
        for (int i = 0; i <= currentLevel; ++i) {
            if (update[i]->forward[i] != current) {
                break;
            }
            update[i]->forward[i] = current->forward[i];
        }
        while (currentLevel > 0 && header->forward[currentLevel] == nullptr) {
            --currentLevel;
        }
        delete current;  // Release the element
    }
}
```

4 Test Results

We implement a random generator to generate skip lists of scale N. And then test performance of random searching, insertion and deletion. We all tested

$$N = 1000,\ 5000,\ 10000,\ 50000,\ 1000000,\ 5000000,\ 1000000$$

, and for each scale, we tested for 10 times, all testing results are drawn in the final figure, and we use average values to fit curves, here are the results:



Graph 2: Time results of three operations

Detailed runtime are attached in the folder. In the fit figure, we can see that skip list has a reallu excellent performace - as the N grows bigger, the runtime does not change much. And except the smallest scale the result fits well to the logarithmic complexity. The abnormal result of the smallest scale can be explained by experimental error. We then test again for some times. And the whold trend remains the same, which can prove that our implementation is in line with our analysis.

5 Conclusion

Skip list is a really good structure in theory. And in this project we test its practical performance to find that it is really fast, just as our imagination.