Segment Skip Lists - An Extension of Skip Lists

Translation compiled by Kirito Feng and Dessertion

Li Jiyang

CTSC 2009

Contents

Abstract

This article focuses on an extension of the skip list - the segment skip list, and related proof of their time complexities. Additionally, the advantages of skip lists and their use in informatics olympiads will be analyzed.

§1 Introduction

Data structures are an important part of informatics competitions. These competitions typically test our ability to maintain and process data. These problems typically require us to maintain an array-like structure, and give us queries that include inserting elements. Common implementations for this include BBSTs, square root decomposition, BITs, etc.

> **Remark.** What type of ungodly terror are Chinese Fenwick trees for them to be included in this sentence...

Skip lists are a new type of data structure that support dictionary lookups, such as insertion, deletion, and lookup from an ordered table. The skip list's time complexity is based on randomization and probability. The expected time complexity of the insert, delete, and find operations are $\mathcal{O}(\log N)$.

At the same time, due to its use of linked lists, it is easy to write and expand upon. It can replace the use of balanced binary search trees for dictionary operations. Because of its simple structure, it does not require tree rotations. This property means that it is able to be used in parallel processing.

> **Remark.** The cited paper, by William Pugh, is entirely about using skip lists for parallel processing. In it, he mentions that it is difficult to perform parallel processing with traditional BBSTs because of the difficultly in performing parallel deletes and rotations.

However, the basic skip list only supports dictionary operations for insertion, deletion, and lookup, and does not maintain interval aggregates. With further analysis, we can find the structure of a binary tree within the skip list. Thus, we can maintain interval information, so that we can perform range queries. This article will explain how to augment the insert and delete methods to maintain interval information.

This extended skip list, also known as a segment skip list, can completely replace BBSTs, as it supports both dictionary lookups, and interval operations. At the same time, it retains its original features, its simplicity, and its ability to support sequential operations.

> **Joke.** If the skip list is the *ideal data structure* for OI, please explain why it's so barely mentioned...

§2 Introduction to Skip Lists

§2.1 From Linked List to Skip List

As a common and flexible data structure, linked lists are often used to maintain a single dimension of data. The linked list can easily perform insertion and deletion operations, but the cost of looking up a number from a linked list can be significant (time complexity $\mathcal{O}(N)$). Conversely, maintaining an ordered array supports a search operation in $\mathcal{O}(\log N)$

time using binary search, but maintaining the sorted property of the array when inserting and deleting elements is quite computationally expensive (time complexity $\mathcal{O}(N)$. If we can maintain an ordered list and support binary searching, we can implement the insertion, searching, and deletion operations (dictionary operations) in $\mathcal{O}(\log N)$ time.

Let us first consider a static linked list.

A simple idea is to add some "accelerated pointers", which skip over several elements, to speed up the search.

We can use an idea similar to square root decomposition to obtain a time complexity of $\mathcal{O}(\sqrt{N})$.

By recursively splitting the array in half with each element, we can obtain a structure similar to a segment tree, and a time complexity of $\mathcal{O}(\log N)$, as we can effectively binary search for the desired element.

However, if we are dealing with a dynamically changing array, we cannot maintain this structure with a reasonable time complexity. Thus we can introduce randomization. We store L linked lists. For each element, we randomly generate a height, h. We insert this element into lists $1, 2, 3, \ldots, h-1, h$ in sorted order. As the number of elements in the ith linked list is greater than the number of elements in the $i+1$th linked list, we can find the correct index by iterating the linked lists $L, L-1, \ldots, 2, 1$, and traversing along the appropriate linked list.

The expected time complexity of this operation is $\mathcal{O}(\log N)$.

§2.2 Formally Defining the Skip List's Structure

The skip list is composed of several linked lists $L_1 L_2, \ldots L_M$, and a descending pointer, which satisfies the following properties:

1. Each list L_i is a sorted linked list, which contains two special elements: $-\infty$, the starting node, and $+\infty$, the terminating node.
2. L_1 contains all elements, and $\forall i, L_{i+1} \subseteq L_i$.
3. For all elements in L_i, there exists a descending pointer to the element in L_{i-1}.

The following are some definitions that will be used for the remainder of the article.

Definition 2.1. A **horizontal edge** is a pointer in a linked list in one of the layers.

Definition 2.2. A **longitudinal edge** is a descending pointer, from one element to its representation in another linked list.

Definition 2.3. A **block** is a single element and its longitudinal edges. Its height equals the number of nodes it contains.

Definition 2.4. The **predecessor node** and **successor node** are the neighbours of the node in L_1.

Definition 2.5. The **underlying node** is defined as the occurrence of a given element in L_1.

Definition 2.6. Path(v) refers to the sequence of visited nodes when searching for the index v in the skip list.

Definition 2.7. RevPath(v) is the sequence Path(v) in reverse order.

§2.3 Dictionary Operations on Skip Lists

§2.3.1 Searching

Searching for a given index is equal to the process of constructing Path(v). The pointer starts from the staring node ($-\infty$) of the top layer. If the next node on the same layer is greater than the desired value, we move down a layer. This process is repeated until it's no longer possible to move to another node, at which point the current node is returned.

§2.3.2 Inserting

The height of the node to be inserted is randomized. In this paper, the probability of getting a height of h equals $\frac{3}{4^h}$. The reason for this specific function will be further explained in the third part of this article.

When inserting, ensure that it is inserted on *every* layer below the given height.

Remark. While this seems silly now, as you can easily achieve a space complexity of $\Theta(N)$, this makes the implementation of segment skip lists, described later on, much simpler.

§2.4 Deleting

Deleting is similar to insertion. Find the node to delete, and then fix all of the linked list pointers.

§3 Structure and Basic Operations of the Segment Skip List

§3.1 Supporting Segment Operations

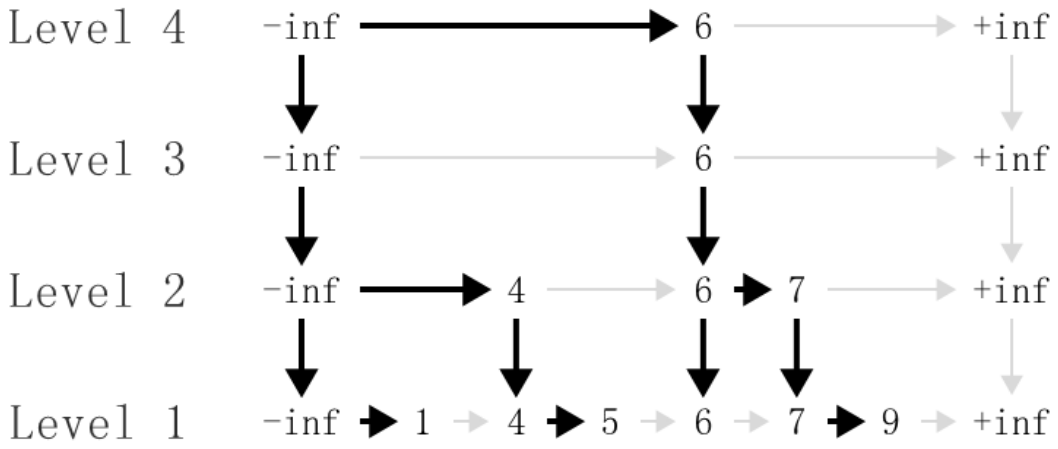
In this section, a segment tree is extracted from the structure of the skip list, forming the segment skip list, which is capable of maintaining interval information.

For a given skip list, we define

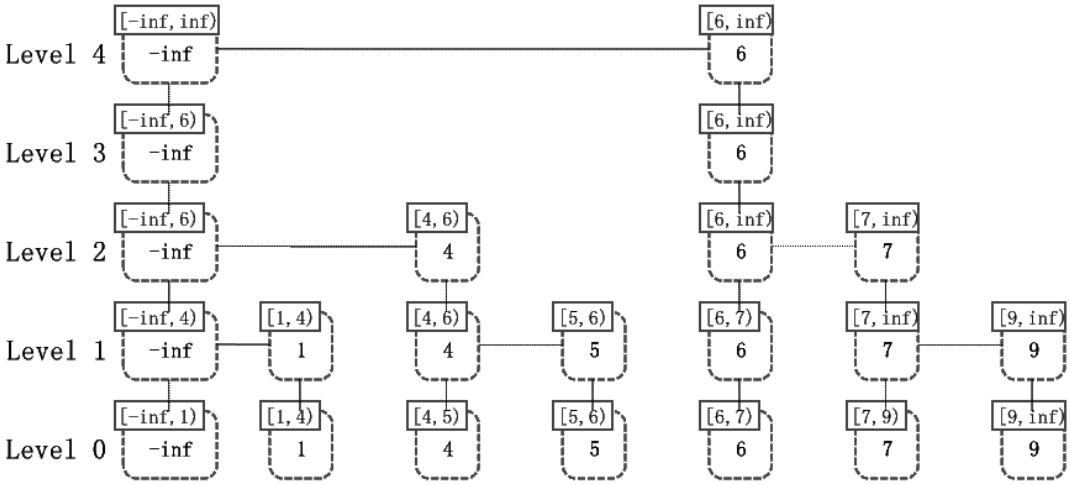
Definition 3.1. A **solid edge** is a horizontal edge that connects to the highest occurrence of an element, or a longitudinal edge.

Definition 3.2. Any edge that is not a solid edge is called a **virtual edge**.

Removing all virtual edges, we obtain a BST structure.



We can already see the segment tree's structure. Furthermore, if we change discrete points to continuous intervals, and add a 0th layer, then we obtain an interval tree.



At this point, we have found the segment skip list.

§3.2 Formalizing the Segment Skip List

The segment skip list is an ordered collection of linked lists $(L_0, L_1, L_2, \ldots, L_N)$ that satisfies the following properties:

1. Each list L_i is monotonically increasing from left to right, and has two special elements; $-\infty$, the starting node, and $+\infty$, the terminating node.
2. L_0 and L_1 contain all of the elements, and for all $i \geq 2$, L_i contains some subset of elements such that $L_N \subseteq L_{N-1} \subseteq \cdots \subseteq L_2 \subseteq L_1 \subseteq L_0$.
3. For all $i \geq 1$, each element in L_i has a pointer to its corresponding element in L_{i-1}.
4. Each node with a value less than $+\infty$ corresponds to an open-closed interval. The left boundary is the index of the node. The right value is equal to the value of the next node along the horizontal edges. This value does not need to be stored as it can be dynamically computed while searching.

The terminology described above is virtually unchanged from 2.2 except for the addition of a 0th layer, representing the underlying node.

This is similar to a segment tree, with a range of $[-\infty, \infty)$.

§3.3 Maintaining the Aggregates

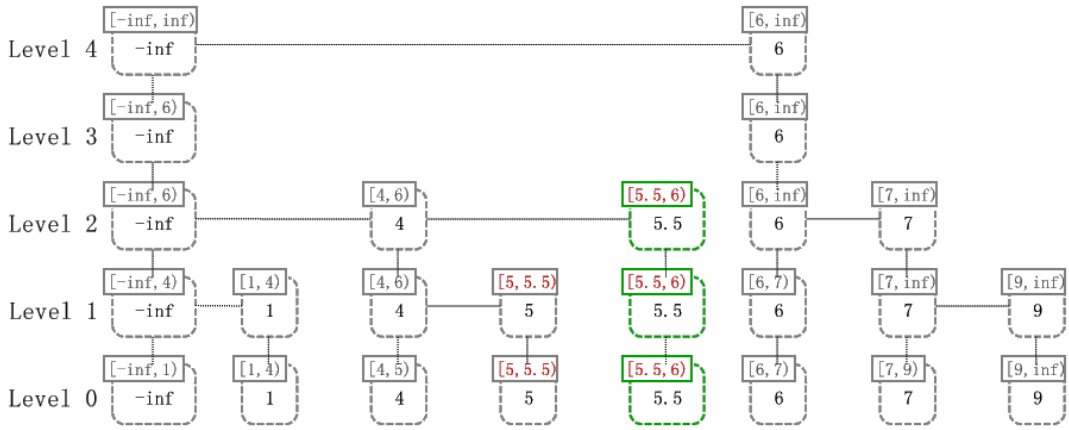
The next step is to maintain information over segments. There are two types of information to maintain.

1. Aggregate over segments: segment min, max, sum, etc. These can usually be computed by merging the aggregates of its children, typically in $\mathcal{O}(1)$ time.
2. Segment information: these are usually segment updates, and can be propagated from a node to its two children. Much like with a segment tree, this information should be stored as far up the tree as possible to reduce the time and space complexity.

These two aggregates have different characteristics and operations. In the next section, we will introduce how to maintain these aggregates when we preform insert and delete operations.

§3.3.1 Insertion

While the structure of the underlying segment tree may change, we can observe that at most $\mathcal{O}(\log N)$ intervals will be affected, as shown by the following diagram.



In the figure above, the nodes with a green outline are added, and the nodes with a red outline are modified. Observe that except for $\text{Path}(v)$ and $\text{Path}(v - \varepsilon)$, the interval corresponding to the nodes does not change.

Therefore we only have to update $\mathcal{O}(\log N)$ nodes.

We perform the following operations when inserting v:

1. Propagate the segment information along $\text{Path}(v)$, so we know what to store at v.
2. Insert v, with the correct segment information.
3. Update the segment aggregates along $\text{Rev-Path}(v)$.
4. Update the segment aggregates along $\text{Rev-Path}(v - \varepsilon)$

The correctness of this algorithm relies on the following two lemmas:

Lemma 3.3

The path contained in Path(v), the interval represented by v, contains v.

Proof. Each time the value retrieved to the node must be less than or equal to the search target, that is, all nodes in Path(v) are less than or equal to v, and each node indicates that the left end (closed) of the interval must be less than or equal to v. For the right endpoint of the interval, it must happen in a certain comparison. It is found that v is less than this value and Path(v) no longer goes to the right but instead downward, so the right endpoint (open) of the interval must be greater than v. □

Lemma 3.4

After inserting v, only the intervals represented by the nodes in Path(v) and Path($v-\varepsilon$) will change.

Proof. Firstly, the index values of the nodes are unchanged, and thus the left endpoint of each interval is unchanged. So you can only consider the right endpoint. By definition, the right endpoint is the index value of the successor node of the last longitudinal edge in the search process. For nodes other than Path(v) and Path($v-\varepsilon$), the horizontal pointer of the starting edge of the longitudinal edge does not change. In summary, after inserting v, only the range of the range represented by the nodes in Path(v) and $Path(v-\varepsilon)$ will change. □

Only the nodes whose interval range changes need to change the interval information and the nodes whose corresponding intervals contain v, that is, only the nodes on Path(v) and Path($v-\varepsilon$).

§3.3.2 Deletion

Deletion is similar to insertion. The general process is as follows:

1. Clean up all segment information along Path(v) and Path($v-\varepsilon$). This means propagating segment information as high up the tree as possible. This is necessary because the range represented by $v-\varepsilon$ will change if v is deleted.
2. If any of v contain segment information, then they represent the left endpoint of segment information, and thus cannot be deleted. Thus we instead have to treat them as non-existent when updating segment aggregates, but cannot delete the node itself.
3. If none of v contain segment information, then we can delete the nodes.
4. Finally, update the segment aggregates for Rev-Path(v) and Rev-Path($v-\varepsilon$).

Note that for some aggregates, you can update both the segment information and the aggregates simultaneously, allowing for a slight optimization.

§4 Analysis of the Advantages and Time Complexities of Skip Lists

§4.1 Time Complexity Analysis of Search, Insertion, and Deletion

First, let $\max lev \to +\infty$. We will first derive the time complexity for search and the expected path length of search. Let $f(N)$ represent the expected search path length for a skip list with N nodes. There are three parts to this computation:

1. The expected search path length for a skip list with heights from 1 to $\max lev$. This is basically equivalent to a skip list with $\mathcal{O}(pN)$ expected path length.
2. A pointer from the second level to the first (in a segment skip list it would be from the first level to the zeroth level).
3. The path length of right-movement on the first level (in a segment skip list it would be the path length of right-movement on the zeroth level). The probability of being able to move right is $1 - p$, which will be denoted as q.

Given this, then:

$$\begin{aligned} f(N) &= f(pN) + 1 + q + q^2 + q^3 + q^4 + q^5 + ... \\ &= f(pN) + \frac{1}{p} \end{aligned}$$

Solving, we get:

$$f(N) = \log_a N, \text{ where } a = \left(\frac{1}{p}\right)^p$$

Time complexities for insertion and deletion are related with search path length, as well as with height; that is, $\mathcal{O}(\log_a N + \max height) = \mathcal{O}(\log_a N)$. When $\max lev$ is a constant (that is, not going to infinity), then $\max height = \max lev$. Due to the nature of the limitations of $\max lev$, it is difficult to analyze the time complexity in these situations, however, in practice there won't be many problems (unless $\max lev$ is set very small). It makes programming simpler and allows fixing $\max lev$ and as such is recommended.

§4.2 Space Complexity Analysis

Like above, set $\max lev \to \infty$

1. Expected height for a single node:

$$\begin{aligned} h_x &= 1 + p + p^2 + p^3 + p^4 + ... \\ &= \frac{1}{1-p} \end{aligned}$$

2. Expected space complexity:

$$\text{Number of nodes} = N \times h_x = \frac{N}{1-p}$$

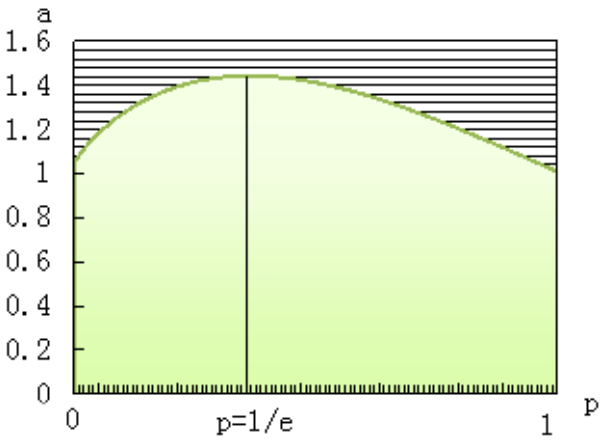
3. Largest max height analysis:

$$P(h_{max} \le m) = (1 - p^{m-1})^N$$

4. When $\max lev$ is a constant (that is, not going to infinity), expected node height is less than or equal to the analyzed value above (i.e. #3); as such, space complexity will also be less than or equal to the analyzed value above.

§4.3 Probability for p

This is the graph for $a = \left(\frac{1}{p}\right)^p$:

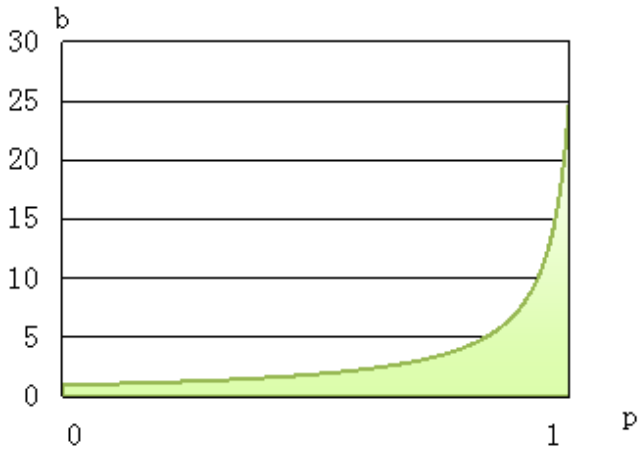


$$f(N) = \log_a N = \frac{\log_2 N}{\log_2 \left(\frac{1}{p}\right)^p}$$

$p = \frac{1}{e}$ gives the best constant factor, with $f(N) \approx 1.88416939 \log_2 N$.
$p = \frac{1}{2}$ and $p = \frac{1}{4}$ both give $f(N) = 2 \log_2 N$.

While $p = \frac{1}{e}$ gives the better constant, the implementation is quite difficult, thus either $p = \frac{1}{2}$ or $\frac{1}{4}$ is preferred. Due to space complexity limitations, $p = \frac{1}{4}$ is more commonly used.

The following is a graph showing the space complexity $b = \frac{1}{1-p}$.



It can be seen that $p = \frac{1}{4}$ is expected to have better space complexity (as it appears to be better than $p = \frac{1}{2}$ by a factor of $\frac{2}{3}$).

Thus the most common value of p is $\frac{1}{4}$.

§4.4 Maximum number of levels (Time complexity regarding max lev)

For a normal skip list, there is no limit for the maximum height for levels (i.e. $\max lev \to \infty$), however, oftentimes it's set to a constant for simplicity of implementations. It's difficult to analyze the time complexity with the introduction of $\max lev$, but in practice it seems to work quite well. One can pick an appropriate constant for $\max lev$ by analyzing the largest height, for example, for $p = \frac{1}{4}, N = 10^6$, one can pick $\max lev = 16$, resulting in $P(h_{max} \leq \max lev) = (1 - p^{\max lev - 1})^N \approx 0.999069111$ (a very good choice).

§4.5 Segment Skip Lists

The operations implemented by segment skip lists are quite similar to that of skip lists, and have a similar time complexity, which is dependent on the expected search path.

The time complexity is still expected to be $\mathcal{O}(\log N)$, but with a larger constant.

As for the space complexity, since it has one more layer than a skip list, it can be expressed as $\mathcal{O}\left(\frac{2-p}{1-p}N\right)$. While this appears to be quite large, when we set $p = \frac{1}{4}$, we obtain $\mathcal{O}\left(\frac{7}{3}N\right)$, which is only slightly worse than a segment tree.

Since each node needs to store aggregates, the constant is slightly larger, however, the overall memory complexity is still expected to be $\mathcal{O}(N)$.

§4.6 Advantages of Skip Lists

Skip lists are essentially a linked-list. Operations are simple and rotations are not required. Skip lists have shorter code than your average BBST implementation, and importantly, the operations are easy to think about and don't require any special tricks/techniques to shorten the code. Additionally, they support parallel operations. Since they don't require rotations, they can be stored on several different media (e.g. disk drives), thus permitting parallel operations.

Skip lists only need to adjust the value of p to modify their space/time complexity tradeoff.

Skip lists do not need to store any additional information to maintain their structure, thus making them very memory efficient.

> **Remark.** Very efficient except for the fact that a node can be stored more than once, because of how levels work...but let's just casually ignore that now shall we?

Skip lists can perform operations on sequences, and special operations related to that.

> **Remark.** No idea what that statement means, not sure its relevant either given how its never expanded upon. As a guess, it most likely refers to maintaining a sequence.

§4.7 Comparison of Segment Skip List, Segment Tree, and BBSTs

Segment trees and BBSTs are commonly used data structures in informatics competitions, and they can be used for many topics. The line segment jump table introduced in this paper obtains a relatively balanced structure via randomization and can support all functions of both segment trees and BBSTs. Compared to the segment tree, it has the advantages of being able to dynamically process information and support online inquiry; compared with BBSTs, it has the advantage of being simpler to write.

> **Joke.** I will race anyone who is insane enough to use a segment skip list to solve NOI '05 B with my treap implementation.
>
> Good luck beating 110 lines / 10 minutes.

In summary, the jump table and its extension - line segment jump table, is an excellent data structure.