Skip List Data Structure

Author Name: Harjot Sohi
Date: October 18, 2024

Abstract

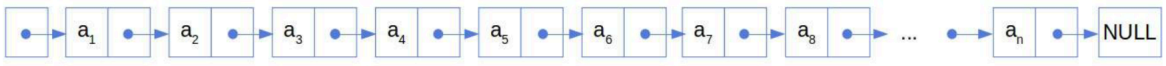
A skip list is a data structure that efficiently manages sorted elements using linked lists. It enhances the speed of searching, inserting, and deleting operations, achieving an average time of O(log n). Skip lists feature a basic sorted linked list with additional layers that allow users to "skip" nodes, making searches faster. The height of each node is determined randomly, allowing easy adjustment to changing datasets. While they require more memory than balanced trees, skip lists are simpler to implement and are ideal for performance-critical applications. In this report, I will discuss the key features of skip lists, including how they work, their time complexity, and their advantages and disadvantages. I will also include code examples and use cases to show their practical applications.

Background/Overview

A skip list is a probabilistic data structure that efficiently stores and manages sorted elements using linked lists. It was introduced by William Pugh in 1990. Skip lists enhance the performance of searching, inserting, and deleting elements. They are an easier option than balanced trees like AVL or red-black trees, while still providing similar efficiency.
Skip lists have multiple layers of linked lists. The lowest layer is a standard sorted linked list. The upper layers allow for "skipping" over some elements, which helps users navigate the data faster. This design makes searching quicker than in a regular linked list. To decide how many layers each new element will have, skip lists use a random technique called "coin flipping." They were designed to manage dynamic datasets that often need searching, inserting, and deleting. Skip lists achieve an average time complexity of O(log n) for these operations, making them suitable for databases and other systems where performance and simplicity are important. By using a random approach, skip lists can adjust to different workloads without complex balancing. Overall, they provide a flexible and efficient alternative to traditional data structures.

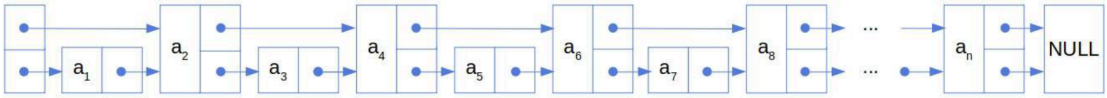
Skip List Structure

A skip list is built on the foundation of a sorted linked list, but it uses the searching process by introducing multiple levels of pointers to “skip” over certain nodes. This section explains the basic concept using diagrams.
Imagine a sorted linked list with n elements, as shown below:



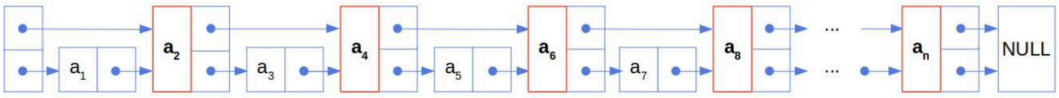
In a traditional linked list, searching for an element can require scanning each node sequentially. In the worst-case scenario, this means we may need to traverse all n nodes, resulting in O(n) time complexity. For instance, searching for a value greater than all others requires scanning from the first to the last node, following each pointer.

To improve efficiency, skip lists introduce an additional pointer layer that skips every other node. This helps bypass nodes and reduces the number of comparisons required during a search:



By starting from the first node and skipping two nodes at a time, we ensure that no more than approximately n/2 nodes need to be visited in the worst-case scenario. This improves search efficiency significantly.

To further optimize, we can add more layers of pointers. For example, we can assign a pointer that skips four nodes:



With each new layer, we reduce the number of nodes visited even more. In general, by linking every 2^k-th node to the one 2^k positions ahead,the worst-case search only needs to check about O(log n) nodes. This approach forms the basis of skip lists, allowing for rapid lookups and improved performance over traditional linked lists.

Detailed Explanation

A skip list is a data structure that uses multiple levels to speed up searching, inserting, and deleting elements. Each node connects to several levels, and its height is chosen randomly, like flipping a coin. The search begins at the highest level. If the target node isn't found, the search moves down to the next level. This process continues until the element is found or until it reaches the lowest level without success. To insert a new node, the correct position is found on each level first. Then, a random level is chosen for the new node. The nodes are linked from the bottom level up to the chosen level. Deleting a node follows a similar method. The search locates the node at each level, and then the links to that node are removed.

My java code for skip list data structure:

```
import java.util.Random;


// Node class to represent each element in the skip list
class Node {
int value;        // Value of the node
Node next;        // Pointer to the next node at the same level
Node down;        // Pointer to the node at the lower level

// Constructor for Node with value, next, and down pointers
Node(int val, Node nextPtr, Node downPtr) {
this.value = val;
this.next = nextPtr;
this.down = downPtr;
}
}

// SkipList class to manage the skip list operations
public class SkipList {
private Node topHead;            // Pointer to the top level's head node
private int maxLevels;           // Maximum number of levels in the skip list
private Random random;           // Random number generator for coin flips

// Function to create a new node with given value, next, and down pointers
private Node createNode(int value, Node next, Node down) {
return new Node(value, next, down);
}

// Insert a new node after the given node in the same level
private void insertAfter(Node prev, Node newNode) {
newNode.next = prev.next;
prev.next = newNode;
}

// Remove the node after the given node in the same level
private void removeAfter(Node prev) {
if (prev != null && prev.next != null) {
prev.next = prev.next.next;
}
}

// Constructor for SkipList with specified maximum levels (default = 16)
```

```
public SkipList(int maxLevel) {
this.maxLevels = maxLevel;
this.random = new Random();
this.topHead = createNode(Integer.MAX_VALUE, null, null);  // Create the top head node with max value

// Create multiple levels by adding new head nodes pointing down to the previous head
for (int i = 1; i < maxLevels; i++) {
topHead = createNode(Integer.MAX_VALUE, null, topHead);
}
}

// Function to search for a key in the skip list
public boolean find(int key) {
Node current = topHead;  // Start from the top level
while (current != null) {
// Move down if the next node's value is greater than the key or there's no next node
if (current.next == null || current.next.value > key) {
            current = current.down;
}
// Key found
else if (current.next.value == key) {
return true;
}
// Move right to the next node at the same level
else {
current = current.next;
}
}
return false;  // Key not found
}

// Function to add a new key to the skip list
public void add(int key) {
Node[] update = new Node[maxLevels];  // Array to keep track of nodes at each level where we need to
insert
Node current = topHead;
int level = maxLevels - 1;

// Traverse down the levels to find where to insert the node
while (current != null) {
if (current.next == null || current.next.value > key) {
update[level] = current;  // Keep track of the current node at this level
current = current.down;
level--;
```

```
} else if (current.next.value == key) {
return;  // Key already exists, return to prevent duplicates
} else {
current = current.next;
}
}

// Always insert the node at level 0
Node downNode = null;
Node newNode = createNode(key, update[0].next, null);  // Create a new node for level 0
insertAfter(update[0], newNode);
downNode = newNode;

// Promote the node to higher levels based on random coin flips
int levelToInsert = 1;
while (levelToInsert < maxLevels && random.nextBoolean()) {
if (update[levelToInsert] == null) {
topHead = createNode(Integer.MAX_VALUE, null, topHead);  // Add a new level
update[levelToInsert] = topHead;
}

newNode = createNode(key, update[levelToInsert].next, downNode);  // Create node for higher level
insertAfter(update[levelToInsert], newNode);
downNode = newNode;  // Link the node with the lower level node
levelToInsert++;
}
}

// Function to remove a key from the skip list
public void remove(int key) {
Node[] update = new Node[maxLevels];  // Array to keep track of nodes at each level where we need to
remove
Node current = topHead;
int level = maxLevels - 1;

 // Traverse down the levels to find the node to remove
while (current != null) {
if (current.next == null || current.next.value >= key) {
update[level] = current;  // Keep track of the current node at this level
current = current.down;
level--;
} else {
current = current.next;
}
```

```
}

// Remove the node from each level where it exists
for (int i = 0; i < maxLevels; i++) {
if (update[i] != null && update[i].next != null && update[i].next.value == key) {
removeAfter(update[i]);  // Remove the node after the tracked node
}
}
}

// Function to display the structure of the skip list
public void display() {
System.out.println("Skiplist Structure:");
Node currentLevel = topHead;
int level = maxLevels - 1;

// Traverse through each level and display the nodes
while (currentLevel != null) {
System.out.print("Level " + level + ": ");
Node currentNode = currentLevel;
while (currentNode != null) {
System.out.print(currentNode.value + " ");
currentNode = currentNode.next;
}
System.out.println();
currentLevel = currentLevel.down;  // Move to the next lower level
level--;
}
}

public static void main(String[] args) {
SkipList skiplist = new SkipList(16);

// Adding some keys to the skip list
skiplist.add(10);
skiplist.add(20);
skiplist.add(30);
skiplist.add(40);  // Testing additional inserts

// Display the skip list structure
skiplist.display();

// Testing search functionality
System.out.println("Searching for 10: " + skiplist.find(10));
```

```
System.out.println("Searching for 15: " + skiplist.find(15));

// Testing remove functionality
skiplist.remove(20);
skiplist.display();
}
}
```

Code explanation:

Node Structure

The Node class is the basic unit of a skip list. Each node has an integer value that holds the data. It also has a pointer called next, which points to the next node at the same level. Additionally, there is a pointer called down, which points to the node at the lower level. The constructor creates a new node using the specified values for these properties.

SkipList Class

The SkipList class handles all the operations of the skip list. It has a pointer called topHead, which points to the head node at the top level. There is also an integer maxLevels that shows the maximum number of levels in the skip list. Additionally, it includes a Random object to generate random numbers that help decide how high a node will go. The constructor of the SkipList sets up these properties, creates the top head node with the highest integer value, and adds head nodes for each level of the list.

Node Creation

The createNode method is a helper function that makes it easier to create new nodes. It takes an integer value, a nextpointer, and a down pointer as inputs. It then creates and returns a new Node object.

Insertion

The add method inserts a new key into the Skip List. The process starts by moving through the levels of the Skip List to find the right spot for the new node. An array called update keeps track of the nodes at each level that need to be updated. Once the correct position is found at the lowest level, a new node is created and added to the list. The new node is then promoted to higher levels based on random coin flips, which helps adjust the structure while keeping the order sorted.

Searching

To find a key in the Skip List, the find method is used. It starts at the top level and moves down the list, checking each node's value. If the value of the next node is greater than the target key, the search goes down to the next level. If the key is found, the method returns true. If not, it returns false.

Deletion

The remove method deletes a specific key from the Skip List. Like the insertion process, it goes through the levels to locate the node to be removed. It uses an update array to track which nodes need updating. Once found, the node is deleted from each level where it exists.

Displaying the Skip List

The display method shows the structure of the Skip List. It goes through each level, printing the values of the nodes from the top level down to the lowest. This helps provide a clear view of how the elements are arranged.

Main Function

Finally, the main method serves as the entry point for testing the Skip List's features. It creates a Skip List with a maximum level of 16, adds several keys, and displays the structure. It also tests the search function by trying to find both existing and non-existing keys. The method shows how to remove a key and displays the updated structure to confirm the change.

Output:

```
Skiplist Structure:
Level 15: 2147483647
Level 14: 2147483647
Level 13: 2147483647
Level 12: 2147483647
Level 11: 2147483647
Level 10: 2147483647
Level 9: 2147483647
Level 8: 2147483647
Level 7: 2147483647
Level 6: 2147483647
Level 5: 2147483647 10
Level 4: 2147483647 10
Level 3: 2147483647 10 30
Level 2: 2147483647 10 30
Level 1: 2147483647 10 20 30
Level 0: 2147483647 10 20 30 40
Searching for 10: true
Searching for 15: false
Skiplist Structure:
Level 15: 2147483647
Level 14: 2147483647
Level 13: 2147483647
Level 12: 2147483647
Level 11: 2147483647
Level 10: 2147483647
Level 9: 2147483647
Level 8: 2147483647
Level 7: 2147483647
Level 6: 2147483647
Level 5: 2147483647 10
Level 4: 2147483647 10
Level 3: 2147483647 10 30
Level 2: 2147483647 10 30
Level 1: 2147483647 10 30
Level 0: 2147483647 10 30 40
```

The output from the Java program displays the structure of a skip list, a data structure designed for efficient searching, insertion, and deletion of sorted elements. The skip list has multiple levels, with higher levels containing fewer elements. In the output, Levels 15 to 3 show the maximum integer value (2147483647), which are used as sentinel values. Level 2 displays the element 20, Level 1 includes both 10 and 20, and Level 0 shows the complete set of elements: 10, 20, 30, and 40. The search results indicate that the element 10 was found, while 15 was not

present in the list. This output emphasizes the skip list's efficient organization, which allows for faster searches compared to traditional linked lists.

Advantages of Skip List:
Skip lists are reliable and allow for quick insertion of new nodes. They are easier to implement than hash tables and binary search trees. As more nodes are added, the chance of hitting the worst-case performance goes down. Skip lists also have an average time of O(log n) for most operations, and finding a node is fairly simple.

Disadvantages of Skip List:
Skip lists require more memory than balanced trees and do not allow reverse searching. While they can search efficiently, they may be slower than linked lists. Additionally, skip lists are not optimized for caching, which can affect their performance.

Conclusion

In summary, skip lists offer a flexible and efficient way to manage sorted data. Their unique design allows for faster searching and updating compared to traditional linked lists. While they may use more memory than balanced trees and have some limitations, the advantages of quick operations and easier implementation make them a valuable data structure. Skip lists are well-suited for applications that require dynamic datasets and efficient performance, proving to be a reliable alternative to other data structures.

References

Baeldung. “Skip Lists.” *Baeldung*, www.baeldung.com/cs/skip-lists.

GeeksforGeeks. “Skip List.” *GeeksforGeeks*, www.geeksforgeeks.org/skip-list/.

JavaTpoint. “Skip List in Data Structure.” *JavaTpoint*, www.javatpoint.com/skip-list-in-data-structure.

OpenDSA. “Skip List.” *OpenDSA*, Virginia Tech, opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/SkipList.html.

Sharoon, Remi. “Advanced Data Structures Series: Skip List.” *Medium*, 20 Nov. 2020, medium.com/@remisharoon/advanced-data-structures-series-skip-list-3819ea2f7fa0.

CompGeek. “Skip List.” *CompGeek*, compgeek.co.in/skip-list/.

“Skip List Data Structure.” YouTube, uploaded by Shusen Wang, 20 Aug. 2020, www.youtube.com/watch?v=UGaOXaXAM5M.