Performance Analysis of LinkedList and SkipList Implementations

Alexandar Djidjev

May 2, 2024

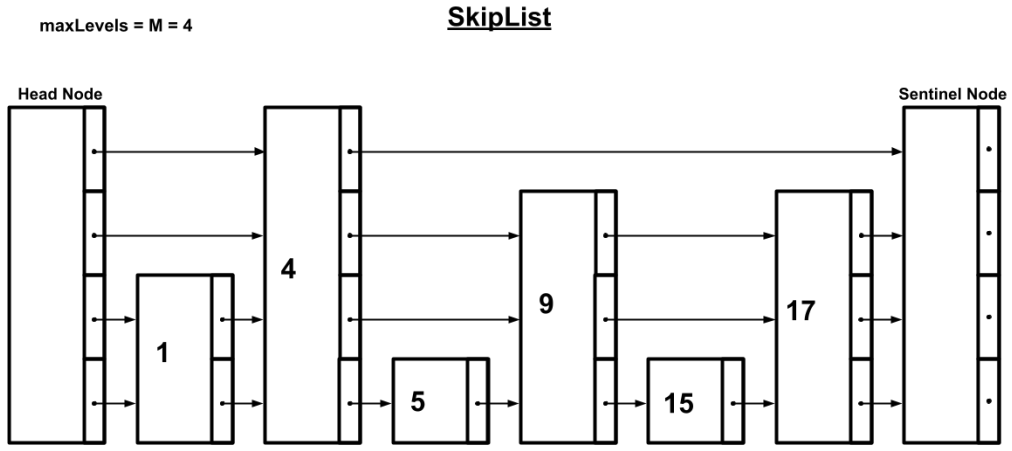
Problem Statement

In this analysis, we will consider the LinkedList and SkipList data structure implementations and analyze the performance of important operations of each list. To demonstrate the differences between these data structures, the scenario of a daily event scheduler application will be used, where each list will store multiple event objects. Each event will have a name and time attribute. The operations performed on the two lists will consist of inserting event objects from a pre-existing dataset in ascending order based on their time of occurrence and then searching for a specified event based, once again, on the time attribute. The insert and find procedures will be analyzed both analytically and empirically with consideration to the time-complexity of the operations. The implementation of the data structures will be in C++ and the empirical testing will use the Valgrind Callgrind profiler.

The LinkedList is a very well-known data structure that utilizes the concept of linking together elements. Every node in the list stores data and a reference, or pointer, to the next node in the link.

Alternatively, SkipLists are a slight modification of LinkedLists because they maintain not only one link of elements, but multiple links on multiple different "levels". Also, SkipLists are probabilistic data structures that always maintain a sorted order. Elements are inserted to their corresponding location based on their value and the number of levels that each node will have is dependent on randomization, or more specifically, coin flips. Each node

will have one level by default, but a coin flip is performed to determine if the node will gain an extra level. Every time the coin lands on heads, the node's level increases. If the coin lands on tails or the maximum number of possible levels is reached (configured in the implementation), the node's level stops increasing. An example SkipList is shown below.



Analytical Analysis

First, we will use mathematical and algorithmic analysis to determine the rate of growth of the runtime functions for each operation. The algorithms used for the each lists' utilized functions are first presented, followed by an explanation and analysis.

LinkedList

The operations used in the LinkedList for this analysis were the insert, find, and insert in ascending order functions. Additionally, the number of operations for the LinkedList's functions were calculated by following a set standard. Instantiating new objects, assigning variables a value, accessing member attributes, and performing logical comparisons are all counted as one operation. Below are the algorithms for each operation and the time complexity analysis for each.

Algorithm 1 LinkedList's Insert

function INSERT($iter, value$)	**Number of Operations**
$temp = new\ LinkedListNode$	2
$temp \rightarrow data = iter \rightarrow data$	3
$temp \rightarrow next = iter \rightarrow next$	3
$iter \rightarrow data = value$	2
$iter \rightarrow next = temp$	2
$increment\ numElements$	1
end function	**Total:** 13

As we can see from the calculation of the runtime function, the number of operations performed is a constant value for any size of the LinkedList. Thus, it is evident that the rate of growth of this runtime function is not dependent on the number of elements in the list which indicates that its rate of growth is $O(1)$.

For this next algorithm, we will consider the best-case and worst-case scenarios, placing an upper and lower bound on the runtime function. The number of times the loop will execute is denoted by k.

Algorithm 2 LinkedList's Insert in Ascending Order

function INSERTASCENDING($value$)	**Number of Operations**
$iter = head$	1
while $iter \rightarrow data < value$ **and** $iter \rightarrow next$!= $null$ **do**	$5k$
$iter = iter \rightarrow next$	2
end while	
$insert(iter, value)$	$O(1)$
end function	**Total:** $2(5k) + 1 + O(1)$

In the best-case scenario, *value* will be less than or equal to the element at the current head of the LinkedList and thus will be inserted at the head. This means that $k = 0$ and the while loop will not execute at all. Thus, the runtime function is $1 + O(1)$ which also has a constant rate of growth, $O(1)$.

Alternatively, in the worst-case scenario, *value* will be greater than all of the existing elements already in the LinkedList and thus, the function will

have to iterate through the entire list, comparing each element with *value*. Therefore, $k = n$ where n is the number of elements currently in the list. The runtime function then becomes $10n + 1 + O(1)$ which has a linear rate of growth, $O(n)$.

The analysis of the LinkedList's find function is similar to the insert in ascending order function since it relies on the linear search method. We will again look at the best and worst-case scenarios. The number of while loop executions is k.

Algorithm 3 LinkedList's Find

function FIND(*value*)	**Number of Operations**
iter = *head*	1
while *iter* → *next* != *null* **do**	$2k$
if *iter* → *data* == *value* **then**	2
return *iter*	
end if	
iter = *iter* → *next*	1
end while	
return *null*	
end function	**Total:** $3(2k) + 1$

Once again, in the best-case scenario, *value* will be the first element in the list and $k = 1$ since the function will return the position of *value* and exit the while loop. Thus, the rate-of-growth is $O(1)$.

For the worst-case scenario however, *value* will be the very last element in the list and $k = n$. The function will have to iterate and check every element in the list. Therefore, the rate of growth is $O(n)$.

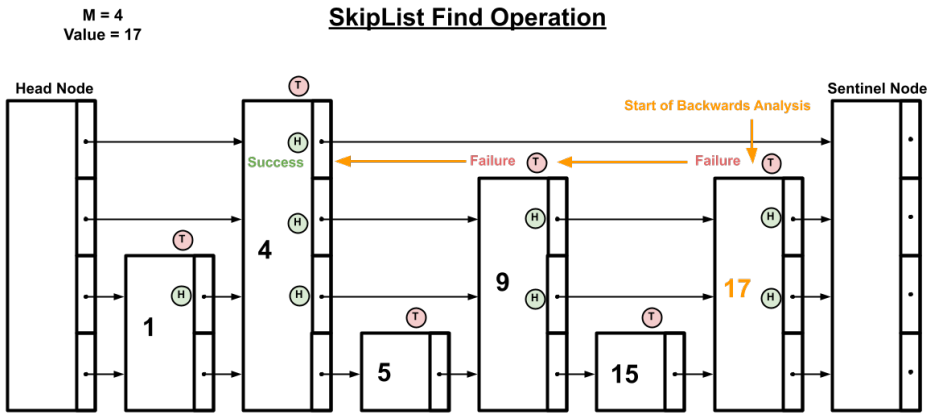
SkipList

Since the SkipList is a probabilistic data structure where we add randomization to choosing the number of levels of a given node, it is best to analyze the average case complexity of both our insert and find functions. Below are the implemented algorithms used in the experiment. Note that the $maxLevels$ used for the empirical analysis was $M = 5$.

Algorithm 4 SkipList's Find

```
function FIND(value)
    node = head
    i = maxLevels − 1
    while i ≥ 0 do
        while node → next[i] < value and node → next[i] → next[i] != null do
            node = node → next[i]
        end while
        decrement i
    end while
    i = 0
    if node → next[i] → data == value and node → next[i] != null then
        return node → next[i]
    else
        return null
    end if
end function
```

The find algorithm relies on the SkipList's multiple level structure and randomized number of levels for each node. According to multiple other sources who have performed analysis of the SkipList's find operation, it is beneficial to work backwards, starting from the located node and working up and to the left through the SkipList structure. As is seen from the diagram below, if we have a SkipList with a maximum possible number of levels, M and if the expected number of leftward steps, $E(S_{left})$ on any given level can be determined, then the expected total number of operations, $E(T)$ (in this case, comparisons) will be $E(T) = E(M \cdot S_{left}) = M \cdot E(S_{left})$. The diagram below visualizes the steps taken to perform this backwards analysis.



The expected number of leftward steps is determined by when the search path goes up. If we look at coin tosses again for determining the number of levels of a given node, a tails corresponds to when the node stopped increasing levels. Thus, if we look at the number of tails on a given level or "toss number", the expected number of tails before we get a heads (on that same "toss number") is the expected number of nodes with the same number of levels before reaching a node with a greater number of levels (when going right to left). This is equivalent to the expected number of leftward steps taken before going up in the search path.

It is important to note here that this expected value applies to SkipLists with an infinite number of elements and infinite number of levels. However, since the implemented SkipLists are never infinite in size or number of levels, the result will give an upper bound to the expected number of total operations for the find function.

The probability distribution applicable to this scenario would be the geometric distribution since it defines a random variable as the number of failures that precede the first success. This is the exact scenario we have, where the failures are considered landing on a tails (or taking leftward steps), and the first success is considered landing on a heads (or moving up in the search path).

S_{left} = number of leftwards steps taken before going up in the search path

p = probability of success (moving up in the search path/landing on a heads)

Therefore, $S_{left} \sim Geo(p)$ and the expected value of a geometric distribution is $E(S_{left}) = (1-p)/p$ with p being any valid probability. The value used in the experimental analysis was $p = 0.5$. Thus,

$$E(T) = E(M \cdot S_{left}) = M \cdot E(S_{left}) = \frac{M(1-p)}{p}$$

Therefore, the upper bound for the expected number of total operations in the find function is proportional to

$$O\left(\frac{M(1-p)}{p}\right)$$

Algorithm 5 SkipList's Insert

```
function INSERT(value)
    increment numElements
    node = head
    i = maxLevels − 1
    predecessorNodes = array[maxLevels]
    while i ≥ 0 do
        while node → next[i] < value and node → next[i] → next[i] != null do
            node = node → next[i]
        end while
        predecessorNodes[i] = node
        decrement i
    end while
    i = 0
    newNode = newSkipListNode
    newNode → data = value
    heads = 1
    while randNum() % 100 < 100 · coinProb and heads < maxLevels do
        increment heads
    end while
    newNode → numLevels = heads
    newNode → next = new array[heads]
    for each i ∈ {0, ..., heads − 1} do
        temp = predecessorNodes[i] → next[i]
        predecessorNodes[i] → next[i] = newNode
        newNode → next[i] = temp
    end for
end function
```

Due to the insert function's reliance on the find operation's functionality, the analysis is quite similar. The first part of inserting a value is finding its correct spot in the sorted order of the list. Then, a randomized "coin flip" process will determine the number of levels of the new node storing the new value. Lastly, every predecessor node to the new node (on every level up to the new node's maximum level) will be updated to point to the new node being inserted. Simultaneously, the new node will be updated (again on every level) to point to each of its predecessor's old next nodes.

Thus, in addition to the expected total number of operations for the find function, the insert function also will execute additional operations proportional to the number of times the *heads* variable is incremented within the "coin flip" while loop.

If we again define a random variable to be the number of heads before the first tails *and* before $heads = maxLevels$ we get

$X =$ number of heads before first tails and before $heads = maxLevels$

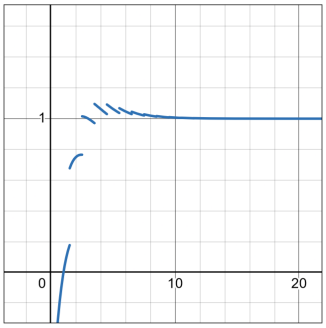
Note that a heads will be defined as a failure and a tails as a success. Let $p =$ probability of failure. Then, $f(x)$ is the probability mass function for the random variable. It is defined as a piecewise function because X is also defined to be the number of heads before $heads = maxLevels$. Thus, it is possible for the levels to stop increasing even if the coin lands on heads due to the *heads* variable reaching $maxLevels$.

$$f(x; p, M) \begin{cases} p(1-p)^x & x \in \{0, \ldots, M-2\} \\ (1-p)^{x+1} & x = M-1 \end{cases}$$

Therefore, the expected value of the pmf function $f(x)$ is

$$E(X; p, M) = (M-1)(1-p)^M + \sum_{x=0}^{M-2} p(1-p)^x$$

As it can be seen from the graph of the expected value function below, for large values of M, the expected number of heads goes to zero. Even for values greater than 2, the expected value is close to 1.



$$E(M) = (M-1)(1-p)^M + \sum_{x=0}^{M-2} p(1-p)^x$$

$$p = 0.5$$

Therefore, to conclude the analysis of the insert function, the expected total number of operations will be the expected total number of operations for the find function plus the expected number of heads before the first tails *and* before $heads = maxLevels$, which is approximately 1 for all values of $maxLevels$, M and $p = 0.5$ (the value used in the experimental analysis).

$$O\left(\frac{M(1-p)}{p}\right) + O(1) = O\left(\frac{M(1-p)}{p}\right)$$

Empirical Analysis

Upon implementing the data structures and their needed functions, the number of instruction reads were obtained for a variety of test input cases. First, there were three different sized pre-existing datasets from where each list then inserted the data while sorting in ascending order. There was a small dataset of 71 events, a medium with 143 events, and a large with 719 events. Furthermore, for every sized dataset, there were three options for the sortedness of the pre-existing file: ascending, descending, and random. The first table shown below, is the instruction reads for the LinkedList's *insertInAscndOrder* function, and for the SkipList's *insert* function.

Inserting Procedure Instruction Reads

	Input File Sortedness					
	Ascending		***Random***		***Descending***	
Size	**LinkedList**	**SkipList**	**LinkedList**	**SkipList**	**LinkedList**	**SkipList**
Small (71)	192,581	136,442	125,214	126,696	42,368	103,793
Medium (143)	701,948	314,917	389,384	276,571	85,352	210,486
Large (719)	16,157,389	3,052,238	8,281,414	1,921,022	429,224	1,048,004

Based on the data collected, it is evident that for most of the input cases, the LinkedList performed more instruction reads than the SkipList. However, when the input file was sorted in descending order, it can be seen that the LinkedList performed faster. This can be attributed to the implementation of the LinkedList's *insertInAscndOrder* function having a constant rate of growth in the best-case scenario (which is when the list reads from a descending ordered dataset and the LinkedList simply inserts the new element at the head of the list each time).

However, when looking at the rate of growth of the instruction reads as the input dataset size increases, it is not a strictly linear growth rate for the LinkedList, nor is it a linear growth proportional to the *maxLevels* for the SkipList. This can be possibly attributed to the lack of a wider range of dataset sizes provided. Conversely, there are clear patterns that we can deduce from the data such as the SkipList's small variability across the different sorted input datasets. This does support the SkipList's find and insert functionality since the important factor is the number of maximum levels and the set probability for increasing a node's levels and not the relative sortedness of the pre-exising dataset.

Next, the find functions for both lists are analyzed. Since, data is automatically sorted in ascending order upon insertion, the dataset only varied with size. Once again, the number of instruction reads were obtained for each test case. Also, it is important to note that for every sized input dataset, the same exact event was searched for in both lists.

Find Procedure Instruction Reads

	List Type	
Size	**LinkedList**	**SkipList**
Small (71)	8,357	1,876
Medium (143)	15,989	2,755
Large (719)	77,045	5,236

As can be seen, the SkipList's find function scales significantly slower than the LinkedList's find function. Once again, the SkipList's innate ability to move faster through the list, due to the higher "fast-lane" levels, and thus compare in a more efficient way, leads to a faster search time. It can be seen again that there is not an apparent linear growth rate for the LinkedList's find operation nor is the SkipList's find rate of growth proportional to M. A wider range of test inputs or a more effective way of tracking instruction operations could lead to different results.

Conclusion

As a result of the analyses, it can be concluded that the SkipList does prove to be faster in its search and insert-in-order operations. The mathematical analysis of the LinkedList's best and worst-case scenario when inserting elements in order was seen to be reflected in the collected data. Furthermore, the SkipList's ability to perform a faster search due to a multi-level structure is also seen from both the data and the algorithmic analysis. The specific growth rates were not clearly visible in the empirical analysis, however future work will focus on improving the variability and range of input datasets for a more comprehensive study.

Resources

While researching the SkipList data structure and performing an analysis, I utilized some very helpful resources to aid my work.

- `https://youtu.be/NDGpsfwAaqo?si=_QLjiQ4p53IqhTLb` - Video analyzing SkipList operations made by Dr. Kevin Buchin from Technical University Dortmund, Germany
- `https://www.math.umd.edu/~immortal/CMSC420/notes/skiplists.pdf` - Lecture notes overviewing SkipLists made by Dr. Justin Wyss-Gallifent from the University of Maryland.